Communication State Transfer for the Mobility of Concurrent Heterogeneous Computing *

Kasidit Chanchio *Xian-He Sun*

Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616
{kasidit, sun}@cs.iit.edu

Abstract

In a dynamic environment, where a process can be migrated from one host to another host, communication state transfer is a key issue of process coordination. This paper presents algorithms for data communication and migration protocols to support communication state transfer in a dynamic, distributed parallel environment. These algorithms collectively preserve the semantics of the communication and support communication state transfer. The assumptions and validity of our solution are discussed and analyzed. Based on our early results in process migration, we implement a prototype system for process state transfer. Experimental results confirm our design is valid and has a true potential in practice.

Index Terms:
Communication Protocol, Process Migration, Distributed and Parallel Processing, Point-to-Point Communication.

1 Introduction

Process migration is a basic function of dynamic programming. 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 environment (heterogeneous process migration). 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]. Process migration can also be used for portability. For example, users can migrate processes from a computing platform to an upgraded one. Process migration

*This work was supported in part by National Science Foundation under NSF grant ASC-9720215, CCR-9972251, and by IIT under the ERIF award.

is a fundamental technique needed for the next generation of internet computation [7]. 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.

SNOW [3] is a distributed environment supporting user-level process migration. Snow provides solutions for three problem domains for transferring computation state, memory state, and communication state of a process, respectively. Methods to transfer execution state of a process are first developed. 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 [8, 9]. Since the selected locations and the augmentation are performed at source code before compilation, the state transfer can be performed across heterogeneous machines. Techniques to transfer memory state are also developed. A graph representation is introduced to model data structures of a process. Methods to transform the structures and their contents into machine independent information and vise versa are provided [10]. Based on our success on computation state and memory state transfer, in this paper we presents a solution to transfer 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. A number of fundamental problems have to be addressed. First, a process should be able to communicate to others from anywhere and at anytime. Process migration could occur during a communication. Mechanisms need to be developed to guarantee correct message arrivals. Second, the problem of updating location information of a migrating process has to be solved. After a process migrates, other processes have to know its new location for future communications. The updating technique should be scalable enough to apply to a large network environment. Third, if a sequence of messages are sent to a migrating process, correct message ordering must be maintained. Finally, 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 the existing fault-tolerant, consistent checkpointing technique. To migrate a process, users can "crash" a process intensionally and restart the process from its last checkpoint on a new machine. Since global consistency is provided by the checkpointing protocol, safe data communication is guaranteed. Projects such as CoCheck [11] follow this approach. On the other hand, mechanisms to maintain safe data communication during process migration can be implemented directly into the data communication protocol. When a process migrates, process migration operations coordinate with data communication operations on other processes for reliability. SNOW, Charlotte [12], and the MPVM project [13] are along the second direction. These systems are message-based and rely on the concept of communication channel. We choose the second direction because the latter is more scalable and less costly than that of the former. Process migration is important enough to receive an efficient mechanism on its own right. Further comparisons are presented in Section 5.

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 to provide 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 operation 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. The operations in process migration are performed mostly at the migrating process, while communication peer processes are only interrupted shortly for the coordination. Second, the process migration protocol is non-blocking i.e., it allows other processes to send messages to the migrating process during the migration. These two properties are quite beneficial for large environments where the number of participating processes is high. Third, 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 PVM (direct communication mode) and TCP. We conduct empirical studies based on a prototype implementation on PVM.

The rest of this paper is organized as follows. Section 2 gives the basic assumptions on the distributed computation model and communication semantics. In Section 3, we discuss the basic idea about communication state transfer and the communication and process migration algorithms. Section 4 shows our experiments by migrating a communicating process in homogeneous and heterogeneous situations. Section 5 discusses related works. Finally, Section 6 summarizes our works and discusses future research.

2 Backgrounds

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.

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 resource for process execution. Second, a number of daemon processes residing on 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 book-keeping 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 [14].

We identify processes in distributed environment in two level of abstractions: *application-level* and *virtual-machine-level*. In the application-level, a process is identified by a *rank* number, a non-negative integer assigned in sequence to every process in a distributed computation [1]. The rank number allows us to make references to a process transparently to its whereabouts. On the other hand, the virtual machine includes location information of a process in its naming scheme. A *virtual-machine-level process identification* (*vmid*) is a coupling of workstation and process identification numbers. They both are non-negative 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 both the scheduler and the daemon do not migrate.

Based on our previous works [8, 10, 3], the migration-supported executable are assumably distributed on potential destination computers prior to process migration. With supervision of the scheduler, a process migration is conducted directly via remote invocation and network data transfers. When a participant in the environment want to migrate a process, it sends a request to the scheduler, which, in turn, decides the destination computer and remotely invokes the migration-supported executable to wait for process state transfer. We name this invocation as *process initialization*. Then, the scheduler sends a migration signal to the migrating process. After the signal is intercepted, the migrating process coordinates the initialized process to transfer its state information. Finally, while the migrating process terminates, execution on the initialized process resumes.

Communication Characteristics

We require that message passing among processes in application-level follows 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 do 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.

Figure 1 shows the protocol stack layout of the communication system for process migration environment. The lowest layer is the OS-supported data communication protocols between computers. The second layer lies communication protocols provided by the virtual machine built on top of the first communication layer. The virtual machine provides three basic communication services. They are the connection-oriented communication utilities, the connectionless communication utilities, and signaling across machines in a distributed environment. We assume the connection-oriented communication to create a bi-directional FIFO communication channel between two processes. In

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

case messages are sent between machines with different platforms, we also assume the protocol in this layer to handle data conversion.

The third layer is the focus of this work. It consists of the migration-supported data communication and process migration protocols which are discussed in the next section in more details. The protocols in this layer provide primitives for the fourth layer, application-level process layer.

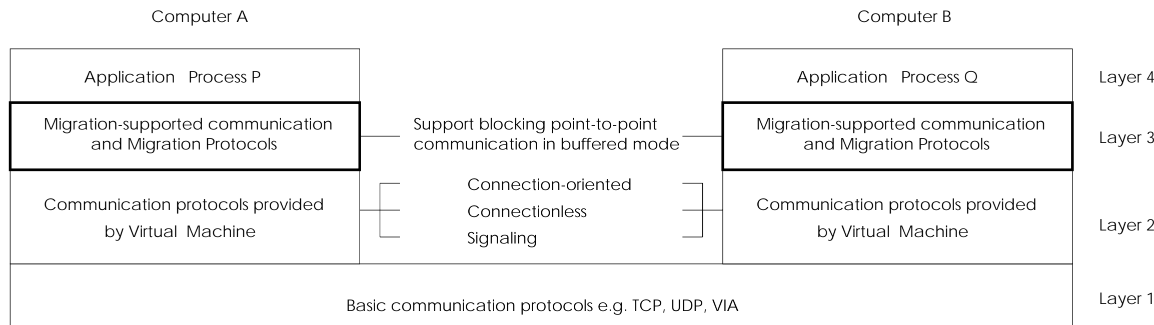

Figure 1. The protocol stack layout.

3 Protocol

This section presents basic ideas of mechanisms to migrate the communication state and algorithms to perform data communication and process migration operations. Since the 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 non-trivial since various communication situations can occur during process migration. In our protocol designs, three basic circumstances are considered.

Capturing and transferring messages in transit

In the first case, to capture messages in transit, processes on both ends of a communication channel have to coordinate each other to receive the messages until none is left. The coordination mechanism is based on the work of Chandy and Lamport [15] 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*. Consequently, the use of received-message-lists effects the design of receive operations. 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.

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 the sender sends a connection request to its intended receiver process.

Having process migration into the picture, the establishment mechanisms must be able to detect the migration (or past occurrence of the migration) and inform the sender process. 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 during 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. During that time if connection requests are intercepted, they would be responded with denials.

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. Based on this scheme, a sender process is not blocked while sending messages to a migrating process. The updating of the receiver's location is also performed "on demand" in the PL table maintained inside the sender's memory space. Thus, the updating is scalable.

Communication state restoration

The scheme for restoring of communication state on a new 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

Based on the previously mentioned conceptual designs, algorithms are invented. The data communication algorithms consists of send and receive algorithms which take care of the connection

```
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;
```

Figure 2. The send algorithm.

establishment and the receive-message-list, while the process migration algorithms consists of two algorithms which run concurrently on the migrating and new processes to carry process migration. The algorithms use the following global variables. *Connected* is a set of rank numbers of connected peer processes. An array *pl* represents the process location table. The *vmid* of process P_i is stored in $pl[i]$.

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, a process which manages process allocation in the environment, to locate the receiver. Once the receiver's location is known, the sender establishes connection and sends messages to the receiver process. Figure 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 Figure 3. The function starts by sending the connection request `con_req` to a receiver process. If the receiver is migrating, it will reject the request and send `conn_nack` back. Then, it will consult the scheduler as mentioned earlier.

The receive algorithm, as shown in Figure 4, is designed to collect messages in an orderly manner in process migration environment. The algorithm stores every messages arrived at a process in the received-message-list in the receiver's memory space.

The receive algorithm also has functionalities to help migrating its peer processes. In case that a process is running the receive event while one of its connected peer process migrates, the receive event may receive a control message from the migrating process. The `peer_migrating` control message is a special message sent from a migrating peer. The message indicates the last message sent from the peer and instructs the closure of communication channel. The reception of this message implies all the messages sent from the migrating process in the communication connection have already been received.

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:          grant_connection_to(p);
8:      else if (receive conn_nack from pl[dest]) then
9:          consult scheduler for exe status and new_vmid of P_dest
10:         if (status = migrate) then
11:             pl[dest] := new_vmid;
12:         else report "error: destination terminated";
13:             return error; end if
14:         end if; end if; end while;
17: return cid;
```

Figure 3. Functions *connect*().

recv (*src*, *m*, *tag*)

```
1:  While (m is not found) 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 con_req then
10:             grant_connection_to(sender of n);
11:         else if n is peer_migrating then
12:             close down the connection with the sender of peer_migrating;
13:     end if; end while;
```

Figure 4. The recv algorithm.

3.2.2 Process Migration Algorithms

The process migration protocol involves algorithms to transfer process state across machines. They are migration and initialization algorithms as shown in Figures 5 and 6, 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 of 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 line 2 and 3 of Figure 5, the migrating process lets the scheduler initializes a process to wait for state transfer before rejecting further connection requests (`con_req`). In rejecting the requests, `con_nack` is responded, causing sender processes to consult the scheduler and redirect their requests to the initialized process. Therefore, before the rejection, the initialized process must already exist and the scheduler has to know its information in hand.

In process coordination, the migrating process sends `disconnection` signal and `peer_migrating` control messages out to all of its connected peers. The `disconnection` signal will invoke an interrupt handler on the peer process if the peer is running computation operations. The handler keeps receiving messages from the communication connection until the `peer_migrating` message is found and then closes the connection. In case the peer process is running a receive event, the receive algorithm may detect `peer_migrating` while waiting for a data message. The peer process then will close down the communication connection by the receive algorithm (see statement 12 of Figure 4).

The last message from a closing peer connection is the end-of-message symbol. The migrating process receives messages from all communication connections to its receive-message-list until all the end-of-message's are received. The migrating process, then, closes the existing communication connections and collects the execution state and memory state. Then, it sends content of the receive-message-list as well as the execution and memory state information to the destination machine. Note that, for heterogeneity, the execution and memory state transfers are based on the techniques presented in our previous works [8, 10] and the XDR encoding/decoding operations performed during data transmission.

On the destination computer, a new process is initialized to wait for process state transfer. Figure 6 shows the initialization algorithm. The initialized process will accept any connection requests from start. At line 2 of Figure 6, the algorithm waits for the contents of the received-message-list from the migrating process. During the wait, if there are any `con_req`'s arrive, the initialized process will grant connection establishment. If the wanted message still do not arrive, the process will also receive new messages and append them to its local 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 the operation at line 2 successes, the received message contents are inserted to the 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. If any

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:     All con_req arrived beyond this point will be rejected;
 5:     Send disconnection signal and peer_migrating
 6:     Receive incoming messages to receive-message-list until getting end-of-messages
 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:     wait for migration_commit msg from scheduler;
12:     cooperate with the virtual machine daemon to make sure that no more
            con_req control messages left to reject;
13:     terminate;
14: end if
```

Figure 5. The migrate() algorithm on the migrating process.

messages or `con_req`'s arrive during the wait, they are responsed by operations similar to those in line 2. The initialized algorithm restores the process state after receiving the state information from the migrating process. Then, it informs the scheduler of migration completion, updates the PL table, and finally resumes program execution.

4 A Case Study: A Parallel Kernel MG Benchmark

To test the proposed event-based data communication and process migration protocols, we have implemented software prototypes and performed experiments on a communication intensive, parallel kernel MG benchmark program. The prototypes consist of a software library for the protocol implementation and its supportive runtime system. As shown in Figure 1, our protocol stack has four layers. For the virtual machine layer, we modify the PVM communication library to accommodate our protocol. PVM has two communication modes. One is direct communication implemented on top of TCP/IP, the other is indirect communication where messages are routed via PVM daemons. We extend the direct communication for our message passing protocol, while we only use the indirect mode for sending control messages. The direct communication establishes TCP connection "on demand" when a pair of `pvm_send` and `pvm_recv` or a pair of `pvm_send` are invoked by the processes on both ends. As shown in Figure 1, the TCP lies in the lowest layer of the protocol stack and has an extended PVM communication library implemented on top. In the second layer, we

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: Restore process state;
6: inform the *scheduler* `restore_complete`;
7: wait for contents of the PL table and *old_vmid* from the scheduler;
8: inform the *scheduler* `migration_commit`;

Figure 6. The initialize() algorithm on the initialized process.

modify the `pvm_send` routine to consult the scheduler when it tries to establish a communication channel with a process that has been migrated. We also add a number of *connectivity service routines* providing utilities for higher-level protocols to request, grant, destroy, and make status reports of every communication channel a process has. We found that the extension only causes small changes to PVM source as most of our protocol designs are implemented in next layer of the protocol stack.

We implement our data communication and process migration protocols in the third layer. Although the send algorithm has most of its operation performed in the extension to the `pvm_send`, operations to access and update the PL table are implemented in this layer. The receive algorithm runs on top of `pvm_recv` and maintains the receive-message-list here. The migration algorithm uses the extended connectivity service routines to coordinate peer processes and disconnect existing communication channel. The initialization algorithm cooperates the migration algorithm and restores process state. Another important programming library in this layer (not shown in Figure 1) contains utilities to collect and restore execution state and memory state of a process. They also handle state transfer directly via TCP. For modularity, we implement such utilities separately from the protocols proposed here. Finally, the migration-enabled process stays in the forth layer.

In our prototypes, the virtual machine and scheduler are employed to monitor and manage runtime environment. We use the PVM virtual machine to handle process creation and termination and to pass control messages and signals between machines. A simple scheduler is implemented to oversee process migration. In current implementation, the scheduler does not support any advanced allocation policy but basic bookkeeping for process migration records.

As a case study, we show here the application of the prototype implementations on the parallel kernel MG benchmark program [16]. The benchmark is written in C and originally runs under 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

with periodic boundary conditions 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 8 processes are used. Since each process has to access data belonging to its neighbors, the data must be distributed to the computation which 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 [16]. Data communication of the MG program is nontrivial. The application exercises extensive interprocess communication; over 48 Mbytes of data on the total of 1472 message transmissions.

We have annotated the program with process migration operations and linked the annotated program to our protocols. In our experimental settings, we generates 8 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 a prior knowledge of the migration incident. Note that no barrier is used to synchronize the processes during a migration.

4.0.3 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. Figure 7 shows a 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 Figure 7. Note that a line between two timeline 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 are 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 the figure, 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 connection to all other processes (due to the original setup of the benchmark), it has to send disconnection signals and `peer_migrating` messages to 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

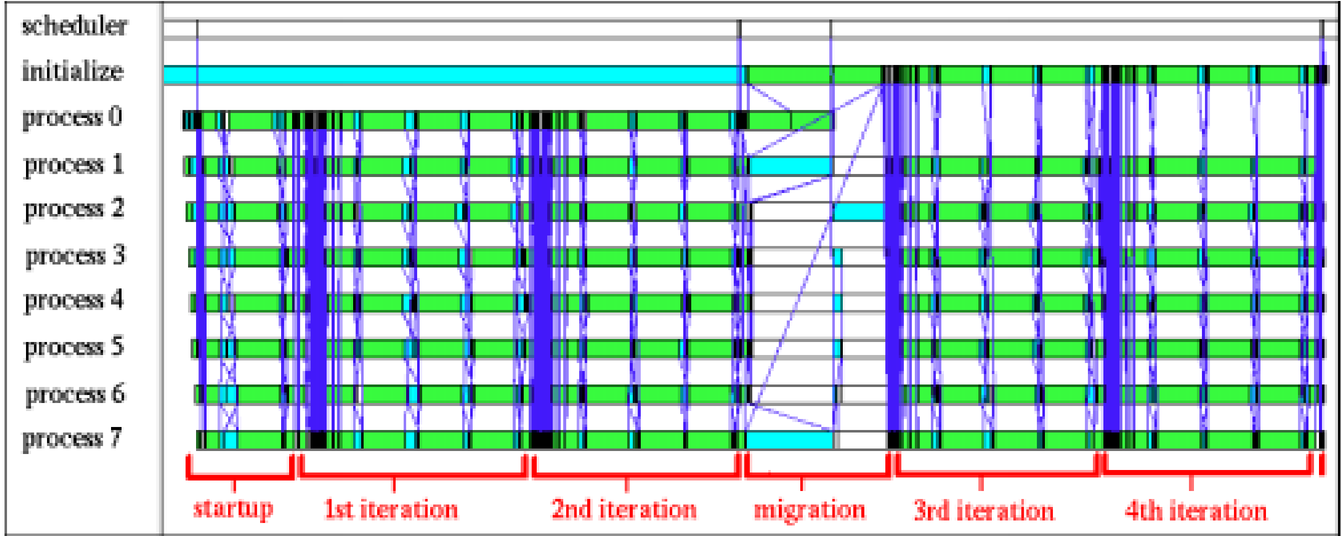

Figure 7. A space-time diagram with a process migration.

existing connection is closed. This operation is shown in area A in Figure 8 [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 Figure 8 shows such execution. In normal operation, the kernel MG process would exchange messages of size 34848 following by 9248 and 2592, etc with its near neighbors. In the area B, some non-migrating processes proceed with the exchanges up to the message size 2592. Then, they have to wait for certain communication to finish before proceeding further until only process 4 can transmit messages of size 800 to its neighbors (area C in Figure 9). Beyond this point, the non-migrating processes have to wait for process 0 to start sending data after the migration finishes.

Finally, following the multigrid algorithm, two messages of size 34848 bytes are sent from process 1 and 7 to process 0 at the start of the third iteration. Since the 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 location of the initialized process for establishing new connections. Such communication are shown by the two lines captured by label D in Figure 9. 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 to the execution and memory state restoration. Since the send data are copied to low level OS buffers, the sender process can proceed with their next execution so that the computation can continue in the area C. The sent data are received after the restoration finishes, resulting XPVM to display two long lines cut across the migration time frame as shown in area D in Figure 9. After that, the migrating process starts

[2] We have performed ten 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.

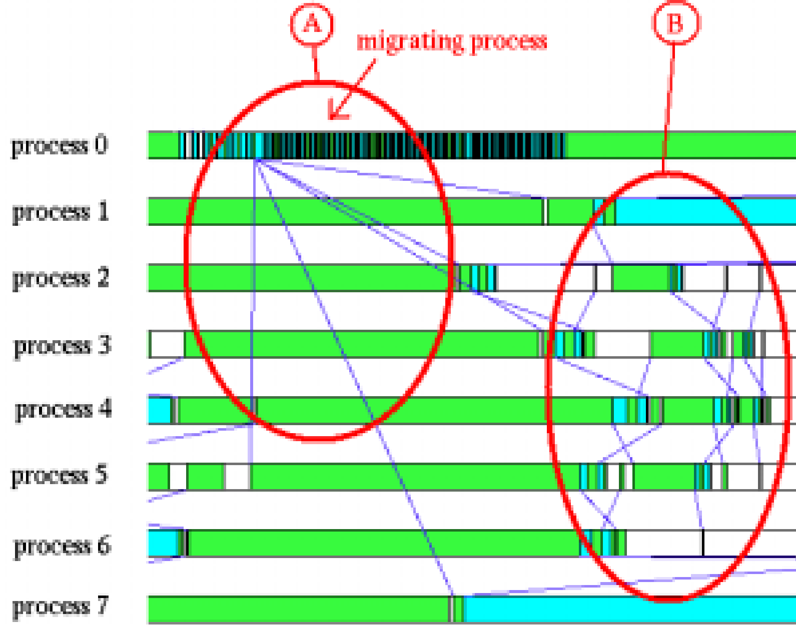

Figure 8. A diagram show the beginning of process migration.

resuming its execution, sends two messages of size 34848 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.

Overheads

In the second experiment, our objective is to exam the overhead 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 are averages of ten measurements. `original` represents the original code running on PVM. `modified` is the migration-enabled process running without a migration. Finally, `migration` represents the migration-enabled process running with a migration.

By comparing the communication time of `modified` and that of `original`, the overhead is evidently small. Although over 48 Mbytes of data on the total of 1472 messages are transmitted during execution, the total overhead of the modified code is only about 0.144 seconds. We believe such 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 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 the 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 details, we find the migration cost to be 2.2922 seconds in average,

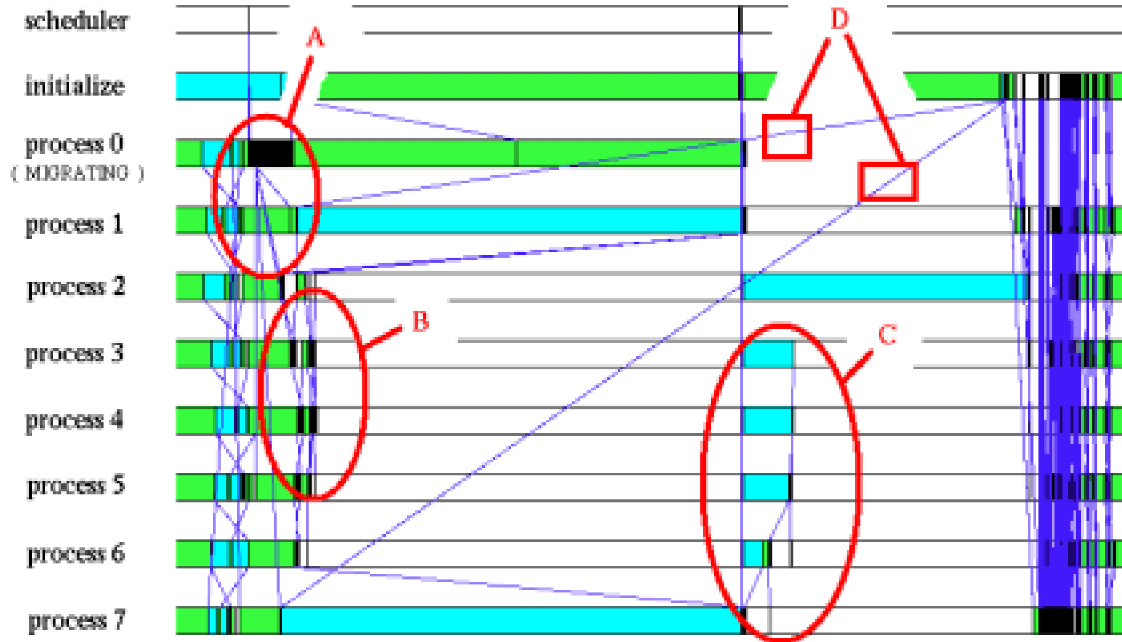

Figure 9. The space-time diagram of a process migration.

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

Table 1. Timing results (in seconds) of the kernel MG program.

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.

5 Related Work

The Chandy and Lamport's algorithm [15] is an early consistent checkpointing algorithm which employs process coordination to achieve global consistency in distributed systems. The CoCheck [11] system implements coordinated checkpointing mechanisms for PVM applications based on the Chandy and Lamport's protocol. Since global consistency is provided, such checkpointing mechanisms can also be used to support process migration. However, we should note forcefully here that the main purpose of systems such as CoCheck is to support fault tolerance not process migration. As a result, their designs suffer two disadvantages which can be alleviated by the specifically optimized systems for migration purpose. First, they require coordination of all processes directly

and indirectly connected to the migrating process. Second, some processes must be blocked from sending messages during checkpointing to maintain consistency.

Alternatively, mechanisms to maintain reliable data communication during process migration can be implemented directly into the message passing protocol. Charlotte [12], MPVM [13], and our SNOW systems are along this direction. These systems are message based and rely on the concept of communication channel. Charlotte supports non-blocking communication and uses the *link* concept, where a process can refer to the same link throughout its computation regardless of mobility. During process migration, the migrating process sends messages along all links to update its location information on the communication peers. In Charlotte's design, messages need not be drained from a link during the migration due to the kernel provided message caching and retransmission mechanisms. Although allowing to write highly concurrent programs, the implementation of kernel-supported communication mechanisms in Charlotte is quite complex [12].

On the other hand, MPVM and SNOW implement reliable message passing mechanisms at user-level and support blocking point-to-point communication. Both systems are implemented on top of PVM. To maintain reliable data communication, the Chandy and Lamport's algorithm is customized. Instead of coordinating all directly and indirectly connected processes, the customized algorithm only performs process coordination to capture communication state between the migrating process and its directly connected peers. MPVM is designed to support transparent process migration for PVM applications. It supports both connection-oriented and connectionless communication based on the PVM direct and indirect communication modes, respectively. A major difference between MPVM and our work is in the connection establishment issue. The MPVM design does *not* maintain automatic connection establishment in point-to-point communications. After the migration, some messages can only be routed via PVM indirect communication, which can severely degrade communication performance. Although both MPVM and SNOW allow other processes to send messages to the migrating process during migration, they employ very different mechanisms. In MPVM, messages are routed to the migrating process via PVM indirect communication. On the other hand, the SNOW protocols transmit such messages directly via a newly established communication channel between the sender and the initialized processes.

The SNOW's protocols are designed to support dynamic distributed environments. Automatic communication establishment is maintained throughout program execution. During a migration, only processes directly connected to the migrating process are coordinated. Our protocols do not block other processes in the system from sending messages to the migrating process. They also allow unconnected processes to make connections and send their data to the migrating process transparently to a migration occurrence. From our experience, the protocols demonstrate simple, yet efficient implementations on top of existing communication software.

6 Summary and Future Works

We have presented algorithms to support communication state transfer in a dynamic, distributed 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 the send algorithm, the sender-initiate technique is implemented so that the sender requests the receiver for a connection. Two vital functionalities are added to the send algorithm to support migration environment. They are the abilities to reconstruct communication channels and to search for the location of a migrated process. In the receive algorithm, we have introduced the receive-message-list as a user-level buffer that keeps messages arrived before their intended receive operations are executed. The algorithm is capable of assisting a migration of a connected peer process by receiving all messages from the communication channel and then closing it down.

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 of heterogeneous process migration for future distributed computation is vital [14]. Works are still left to be done 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 SNOW project [3], develop a compilation system to support semi-automatic process migration. We believe that the development of such tools will advocate new applications of dynamic programming to distributed network computing.

References

[1] D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process Migration," tech. rep., TOG Research Institute, Dec. 1996.

[2] P. Smith and N. Hutchinson, "Heterogeneous process migration : The TUI system," Tech. Rep. 96-04, University of British Columbia, Department of Computer Science, Feb. 1996.

[3] X.-H. Sun, V. K. Niak, and K. Chanchio, "A Coordinated Approach for Process Migration in Heterogeneous Environments," in *1999 SIAM Parallel Processing Conference*, Mar. 1999.

[4] S. Leutenegger and X.-H. Sun, "Limitations of cycle stealing of parallel processing on a network of homogeneous workstations," *Journal of Parallel and Distributed Computing*, no. 3, 1997.

[5] M. Harchol-Balter and A. Downey, "Exploiting Process Lifetime Distribution for Dynamic Load Balancing," *ACM Transactions on Computer Systems*, vol. 15, 1997.

[6] P. Krueger and M. Livny, "A Comparison of Preemtive and Non-Preemtive Load Balancing," in *Proceedings of the 8th International Conference on Distributed Computing Systems*, pp. 336–343, 1988.

[7] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *International Journal Supercomputer Applications*, vol. 11, no. 2, pp. 115 – 128, 1997.

[8] K. Chanchio and X.-H. Sun, "MpPVM: A software system for non–dedicated heterogeneous computing," in *Proceeding of 1996 International Conference on Parallel Processing*, Aug. 1996.

[9] K. Chanchio and X.-H. Sun, "Efficient process migration for parallel processing on non–dedicated network of workstations," Tech. Rep. 96-74, NASA Langley Research Center, ICASE, 1996.

[10] K. Chanchio and X.-H. Sun, "Memory space representation for heterogeneous networked process migration," in *12th International Parallel Processing Symposium*, Mar. 1998.

[11] G. Stellner, "Consistent checkpoints of PVM applications." Proceeding of the 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 Transactions on Software Engineering*, vol. 15, no. 6, pp. 676–685, 1989.

[13] 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.

[14] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[15] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed system," *ACM Transactions on Computer Systems*, pp. 63 – 75, 1987.

[16] S. White, A. Alund, and V. S. Sunderam, "Performance of the nas parallel benchmarks on pvm based networks," Tech. Rep. RNR-94-008, Emory University, Department of Mathematics and Computer Science, May 1994.