Proceedings of the 1993 Winter Simulation Conference
G.W. Evans, M. Mollaghasemi, E.C. Russell, W.E. Biles (eds.)

PROCESS MOBILITY IN DISTRIBUTED-MEMORY SIMULATION SYSTEMS

Janche Sang
Edward Mascarenhas
Vernon Rego
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

ABSTRACT

Our focus is on the novel use of a process-oriented methodology in distributed-memory simulation systems. To the best of our knowledge, the few existing systems which adopt a process-view strictly use message-passing to effect process-interaction in distributed-memory settings. As a result, these systems avoid scenarios in which processes access passive but shared components. This can restrict the manner in which a system is modelled and hinder the phase of distributed model construction. In this paper, we propose an approach which utilizes mobile processes in distributed-memory simulation systems. Mobile processes can move around the system at will, with easy access to remote system components. The approach basically entails the migration of a requesting process with its timestamp to the remote site hosting the requested passive object. Major advantages of this approach include one-time transmission, fixed communication topology, and increased locality of reference. Early results based on lightweight processes show that the mobile process paradigm can be as efficient as the message-passing paradigm.

1 INTRODUCTION

Any given dynamic system generally lends itself to more than a single modelling view. For example, in a queueing network, each job can be modelled as a dynamic process and each server as a passive object. The function of each customer process is to access a server object for service, possibly requiring it to enqueue itself in some waiting queue until the server object is free to provide this service. Alternatively, each server can be modelled as a process, and each customer as an object. The function of the server process is to provide service to each static customer object, enqueueing customer objects other than the one currently being served in a waiting queue.

In a process-oriented simulation system[Franta 1977], *active* components are modelled by *processes*, while *passive* components are modelled by (possibly predefined) data *objects*. As explained in the example above, deciding what role a passive or active system component plays in a model depends solely on the modelling view. This role can change if the view changes, but such a change will usually entail nontrivial code modification in a simulator.

It is worth emphasizing that neither of the two modelling views described above is clearly preferable to the other. However, it is our experience that the mapping of active system processes to mobile simulation processes can potentially enhance an entire modeling effort, from model design and validation, to code debugging and maintenance. Examples of such processes include the movement of sharks and fish over the domain of the ocean, moving automobiles on a problem domain involving streets and traffic lights, moving military tanks over a rugged combat domain, etc.

As discussed in [Franta 1977], the two simulation views described above can be realized through a *mutual exclusion mechanism* and *queueing mechanism*, respectively. Most sequential simulation systems (e.g., SIMULA [Birtwistle, et al. 1979], CSIM [Schwetman 1986]) provide users with a mechanism supporting at least one of these views. We are unaware of any existing parallel/distributed simulation system (e.g., Maisie [Bagrodia and Liao 1990], ModSim [West and Mullarney 1988]) which flexibly supports both views. We believe that these systems effect process synchronization through message-passing (a queueing mechanism). As a result, a user must initially distribute cooperating processes across processors in order to obtain speedup.

Our approach is to distribute passive objects across processors, guaranteeing all processes easy access to these objects. Naturally, a problem arises when a process executing on some processor requires access to an object that is not located on the same processor. For example, a customer process hosted by processor A may require access to a resource or facility object situated on processor B. There are two solutions to this problem. One solution requires implementation of the *Distributed Shared Memory*(DSM) [Libes 1985, Stumm and Zhou 1990] paradigm: the request is transfered to the site on which the passive object is located. The other solution requires process migration: move the requesting process, with its timestamp, to the site on which the passive object is located.

We will show that the DSM approach requires send-reply round-trip message transmissions in order to keep shared object state consistent. In addition, the DSM approach complicates the communication topology, increasing the number of interprocessor links and effectively reducing potential parallelism. One-time transmission, fixed communication topology and increased locality of consecutive accesses makes the Mobile Process approach very attractive.

In this paper we discuss design and implementation issues as they relate to Process Mobility and DSM. We do this using a parallel version of the *Si* experimental simulation system [Sang, et al. 1993b], which is based on a lightweight process library. For completeness and for the sake of comparison, we have also implemented the message-passing mechanism. Our results show that mobile processes are effective, performing as efficiently as the message-passing mechanism, but with slightly higher overhead. Considering the benefits of using mobile processes, this will give analysts a powerful alternative framework for simulator construction.

In order to remove any ambiguity in the definition of the problem that we attack here, we must first distinguish our use of the term *Process* from the commonly-used notion of a *Logical Process* (usually abbreviated as LP) in distributed simulation terminology. The latter term signifies a process with its own local simulation clock and its own input and output communication channels for interaction with other LPs [Chandy and Misra 1979]. Thus, each LP may consist of one or more *Processes* and one or more passive objects. Some attention has been given to the problem of migrating LPs across processors for performance enhancement related to load balancing [Glazer and Tropper 1993].

The remainder of the paper is organized as follows. In Section 2 we briefly describe the organization of the sequential *Si* simulation system. Section 3 details design rationale and implementation issues relating to the parallel *Si* system. Early performance measurements are given in Section 4. We compare the performance of simulations using process migration for remote object access to distributed shared memory techniques for the same remote object access. In addition, we also experiment with process migration and messaging passing on two queueing examples. A brief conclusion is presented in Section 6.

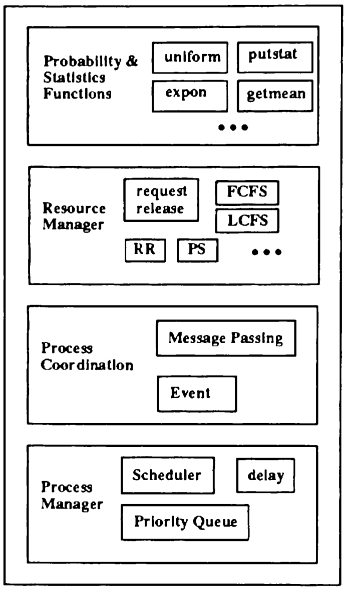

Figure 1: The modules in *Si*

2 THE *Si* SIMULATION SYSTEM

Si, a system for *S*imulating process *i*nteractions, is a process-oriented discrete-event simulation system. It was designed as a research testbed for investigating methods which support simulations efficiently [Sang, et al. 1993a]. It enhances the capabilities of the C programming language through a set of primitives which provide a quasi-parallel programming environment. The *Si* system consists of four major modules: a process management module, a process coordination module, a resource management module, and a statistics module. Figure 1 depicts the layout of modules and examples of functions provided within each.

The process creation task in *Si* is achieved through an invocation of the function *si_create(func, attr, nargs, arg1,· · ·, argn)*. This function returns the process identifier. In the process management module, another important function in *Si* is the *delay()* function. When an executing process decides to suspend itself for *t* units of simulated time, it invokes function *delay(t)*. This function inserts the invoking process's reactivation record, containing its reactivation instant *clock + t*, into the simulation event calendar. Since the invoking process must now undergo suspension, the process scheduler must select as the next process to execute that process in the event calendar whose reactivation time is smallest.

The *Si* system provides two distinct coordination mechanisms to support synchronization between processes. One mechanism is through user-declared events, effected by calling *wait_event()* and *set_event()* primitives. A process is suspended if it invokes function *wait_event(e)* because it is forced to wait until event e occurs. Event e is said to occur when some other process invokes function *set_event(e)*. At this point, all processes waiting for event e are reactivated simultaneously. In *Si*, an event e is declared to be of type *Event* and initialized by the *create_event( )* function.

The other mechanism for process synchronization is through message-passing. Messages can be sent and received with the aid of functions *si_send( )* and *si_receive( )*, respectively. The function *send(tid,msg)* deposits message *msg* in the mailbox of process *tid*. If a process awaits the arrival of a message, *si_send( )* enables the process to access the message and consequently be reactivated. The reverse function *si_receive(&msg)* allows a process to obtain the message from its mailbox. If no message is available, the invoking process is suspended until a message arrives. For simplicity, the size of a message is limited to one word (i.e., the size of an integer or pointer). Note that messages are processed in a FIFO (first-in first-out) order, realizing the queueing mechanism mentioned in Section 1.

Processes are used to model active components of a system. In contrast, facilities are objects used to model passive system components with mutually exclusive access. In other words, processes are dynamic entities which can request access to static facility entities, use these facilities for a time period, and eventually release them to proceed with different activities. *Si* supports two basic functions for facility access: the *request(f)* and *release(f)* functions. The facility object is declared to be of type *Facility* and initialized by an *init_facility( )* function. When a facility f is occupied, other requesting processes must wait in a queue associated with facility f. When facility f is released, a suspended (and enqueued) process is given permission to resume, with access to f.

To illustrate some of the features of the *Si* system, we present a simple example describing a tandem queue. An N-stage tandem queue is a queueing system consisting of N queues in series. A job completing service at server i, $1 \leq i \leq N-1$, will proceed to server $i+1$ and exit the system upon completing service at the last server. In the example shown in Figure 2, 10^4 jobs are made to traverse a system with five queues in tandem. Process *gen_job( )* generates a sequence of jobs. It uses the *expon( )* function to compute interarrival times, and the *delay( )* primitive to space arrival events out in simulated time. Functions *request(f)* and *release(f)* maintain a FIFO ordering in giving jobs access to facilities.

In an alternate view, each station can be modelled by a process, and jobs can be modelled with the aid of "messages". Figure 3 shows *Si* code implementing this view. Each process uses the function *si_send( )* and *si_recv( )* to set up a pipline for processing jobs.

```
#include <si.h>
#define NMAX 10000 /* no. of simulated jobs */
#define IM 5.0 /* mean job interarrival time */
#define SM 4.0 /* mean job service time */
#define N 5 /* no. of nodes in the tandem queue */
Facility f[N]; /* passive components */
Event done;
si()
{ int i;
     for(i=0;i<N;i++) init_facility(&f[i]);
     si_create(gen_cust,NULL,0);
     wait_event(&done);
     ··· output statistical results ···
}
gen_job()
{ int k;
     for(k=0; ; k++) {
          delay(expon(IM));
          si_create(job,NULL,1,k);
     }
}
job(k)
{ int i;
     for(i=0; i< N; i++) {
          request(f[i]);
          delay(expon(SM)); /* being served */
          release(f[i]);
     }
     if(k==NMAX) set_event(&done);
}
```

Figure 2: A Tandem Queueing Model

```
xthread_t sid[N];
si()
{ int i;
     for(i=0;i<N;i++) sid[i]=si_create(server,NULL,1,i);
     si_create(gen_job,NULL,0);
     wait_event(&done);
     ··· output statistical results ···
}
gen_job()
{    ··· use si_send(sid[0],msg) to generate jobs.
}
server(k)
{ int i, msg;
     for(i=0; ; i++) {
          si_recv(&msg); /* a job arrives */
          delay(expon(SM)); /* serving */
          si_send(sid[k+1],msg);
          if(k==N-1 && i==NMAX) set_event(&done);
     }
}
```

Figure 3: A Tandem Queueing Model: an alternate view

3 The PARALLEL *Si* SYSTEM

3.1 The Distributed Multi-Threaded Environment

The Parallel *Si* system employs the Xthreads library [Sang, et al. 1993c] as its kernel. Xthreads is a simple and efficient thread library which has been developed and ported to the nCUBE2 and iPSC/860 hardware multiprocessors. This library supports logical concurrency within each node and true parallelism across nodes in distributed-memory multiprocessors. That is, an Xthreads program consists of a set of processes, cooperating and communicating through the interprocessor network. Processes begin with the same program image and initialize the Xthreads environment. Multiple threads of control sharing a single address space exist within a process.

The Xthreads library supports the notion of thread migration[Sang and Rego 1993]. That is, a thread's context and stack can be transported from one processor to another in a way that allows it to resume execution on the latter at a point of suspension on the former. Description of a similar multi-threaded system can be found in [Chase, et al. 1989].

3.2 Design Rationale

In our distributed model, an LP is a UNIX-like heavyweight process. Each LP has its own local simulated clock and an event calendar. Since an LP contains within itself a set of processes and shared objects, its event calendar is made to store reactivation records for its processes. Further, each LP is associated with input and output communication channels for interaction with other LPs. Various distributed (simulation) algorithms can be used to synchronize LPs. We adopt a conservative algorithm with null messages [Chandy and Misra 1979]. However, because *Si*'s modular design philosophy unambiguously defines interfaces to the system's functional components, almost effortless experimentation with other algorithms [Chandy and Misra 1981, Jefferson 1985] is possible. The only requirement is the simple need to match interfaces.

In contrast to the traditional distributed computing model which distributes processes across processors, our model distributes shared objects across processors. With the provision of a *global-addressing* scheme, processes may be totally oblivious of where the objects they access are located. Therefore, an analyst may use sequential descriptions of models to obtain parallel simulation models.

Both the process-migration methodology and the distributed shared memory methodology may make use of the global-addressing scheme mentioned above. We will discuss the ideas behind both approaches and the advantages and disadvantages of each.

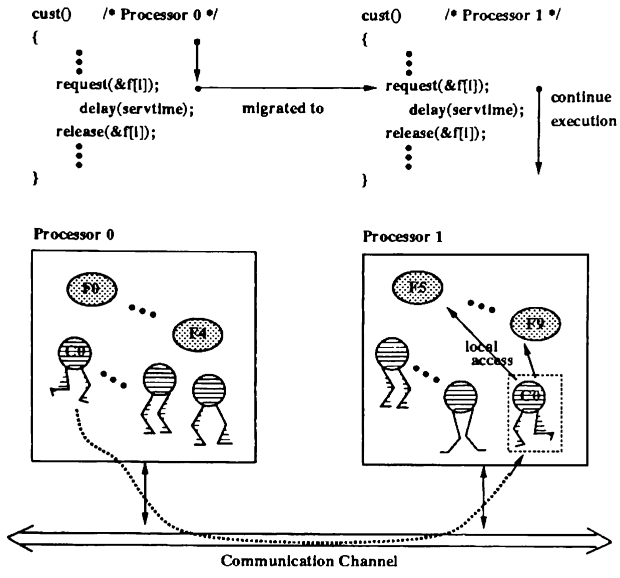
```
cust()   /* Processor 0 */          cust()   /* Processor 1 */
{                                   {
  :                                   :
request(&f[i]);                     request(&f[i]);
  delay(servtime);                    delay(servtime);
release(&f[i]);                     release(&f[i]);
  :                                   :
}                                   }
```

Figure 4: Migrating a process to the remote site

The Mobile Process approach

The Mobile Process methodology we investigate provides a global-addressing mechanism for process migration. Through this mechanism, a process with an access request to a (passive) shared object can migrate to the site hosting this object. Besides information containing the state of the process, the current value of the local simulation clock (i.e., timestamp) must also be sent along with the migrating process. A remote host which accepts a migrant process will allow the process to resume execution when its local simulation clock reaches the timestamp value of the process. The migrant process then accesses the passive object locally and continues execution transparently, as if on its original host.

Note that if a conservative algorithm is used, it is not possible for a host with simulation clock T to receive a migrant with timestamp less than T. With a conservative algorithm, each host may execute a next event e from its list of candidate events only after it is guaranteed that it will not receive an event with timestamp smaller than the timestamp of event e from any other host. This is accomplished through the use of a *lookahead* mechanism. A host receiving a migrant process from another host will not allow its local clock to pass this lookahead value.

One of the advantages of the Mobile Process approach is the need for only a one-time transmission of a message. Once migration to a new host is complete, a process maintains no relationship with its original host. Its reactivation record will now be part of the new host's event calendar. Another significant advantage is that process migration allows the original communication topology of the problem to remain unchanged. For example, in using Mobile Processes to model a tandem queue, the communication topology remains a chain with directed links as shown in Figure 5(a).

Enforcing locality of access through process migration makes the two advantages described above even more significant. Consider, for example, the situation where a process on some host attempts to make consecutive access requests to an object on some remote host. Such a situation can be seen in Figure 4, where we assume that customer *C0* makes a series of access requests to facilities *F5*, $\cdots$, *F9* located on a remote host after it leaves facility *F4*. Migrating customer *C0* to the remote host will reduce the cost of communication since the series of access requests will now be made locally instead of remotely.

A disadvantage of the Mobile Process approach is that there may sometimes be a large amount of state that needs to be moved. This will increase the cost of process migration. Fortunately, it is possible to control the amount of state and stack sizes by refraining from the use of a large number of local variables and access to shared variables in highly nested function calls. It is also possible to develop more sophisticated schemes to optimize or reduce the amount of state that must be moved from one host to another.

The DSM approach

An alternate strategy for the support of global-addressing involves techniques used in the implementation of a shared memory facility on a distributed memory environment. One such technique is to forward a request, along with its timestamp, to the appropriate remote host. At first glance, this approach would appear to be more efficient than process migration simply because the of low message transmission costs. However, a closer examination reveals that efficiency depends on more than just a comparison of transmission costs.

Consider the situation in which a process makes a request to a facility located on a remote host. Once a message containing this request is sent to the remote host, the requesting process must block pending receipt of an acknowledgement. Consequently, round-trip message passing is required. In addition, the sending host must add the receiving host's id to its input channel list. Otherwise, the sending host would have no way of receiving an acknowledgement. Without the need for an acknowledgement, a violation of causality would ensue following the continued execution of a process that does not block after sending an access request. Thus, the method requires that links which would otherwise be uni-directional now be made bi-directional. Also, because the communication topology may change, processes cannot be migrated once created.

To make a simple comparison with the Mobile Process approach, we use the same tandem queue model. Assume that processes (i.e., customers) are created on processor *P0*, and shared objects (i.e. servers) are distributed across processors *P0*, $\cdots$, *P3*. Since all processes will access these shared objects, all of the processors *P1, P2, P3* will have to communicate with processor *P0* (see Figure 5 (b)). If we scatter the processes to be created among the four processors, the communication topology becomes a complete graph (see Figure 5 (c)).

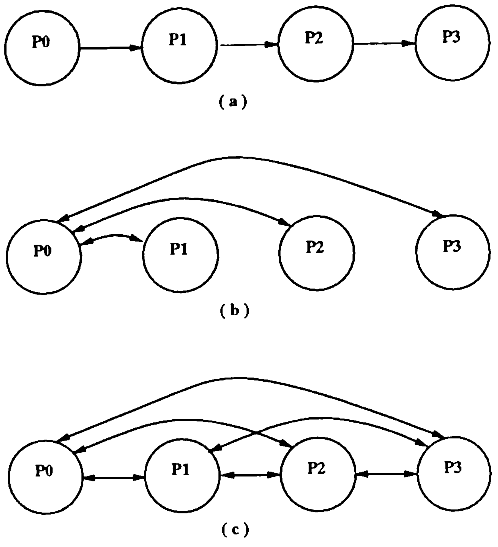

Figure 5: The Communication Topology

3.3 Extensions to the User Interface

One of our objectives in the design of the parallel *St* system was to retain as much of the simple user interface as possible. With the provision of a global-addressing environment, user interface modification is limited to the distribution of shared objects. To assign objects to processors, we extended the user interface by providing more information via function parameters. For example, the facility initialization procedure was modified to look like init_facility(&f,pno) where *pno* is the host at which the facility *f* resides.

Note that initialization statements should be executed globally, by all processors. In this way, each host knows where a particular facility is located so a process may be migrated to this facility when necessary. The following example shows how facilities are distributed evenly among processors.

```
for(i=0; i<num_facility; i++)
    init_facility(&f,i*num_nodes/num_facility);
```

```
request(f)
{
    if (facility f is non-local)
        migrate the calling process to the site where f is
            located;
    /* now f is local to the requesting process. */
    perform operations as in the sequential version;
}
release(f)
{
··· modified in the same way as in request(f)
}
```

Figure 6: The modified functions request( ) and release( ) using process migration

3.4 Modifications to System Functions

Some functions, such as *request(f)* and *release(f)*, were modified to allow for remote facility access. Figure 6 shows the pseudo-code required for handling a facility. The function *request(f)* first determines whether facility *f* is located on the local host (i.e., the one on which the access request is made) or located on a different host. If *f* is local, then action is taken in accordance with the sequential version of *St*. Otherwise, the process making the access request is migrated to the remote host. Observe that when the migrant process resumes execution on its new host, it resumes execution immediately following the *if* statement. That is, it attempts to ascertain the status of the facility. If the facility is occupied, process execution must be suspended. The process is made to wait in a queue at the facility until the facility is free to respond to its request.

A different implementation, using DSM approach, is illustrated in Figure 7. In the function *request(f)*, a request message is sent if the facility *f* is determined to be non-local. The sending process then awaits an acknowledgement. Note that the LP hosting the sender can switch control to another process if and only if such a switch does not alter simulation logic. A server process *fac_server( )* on each processor hosting a facility *f* undertakes the task of accepting and processing request messages. On receiving such a request, each server responds with an ACK once the request is satisfied successfully. If the facility is occupied when a request arrives, the server enqueues the request in a queue at the facility.

4 PERFORMANCE MEASUREMENTS

The parallel *St* system has been implemented successfully on the nCUBE2 hypercube. A port to the iPSC860 hypercube is under way. Both machines are distributed-memory multiprocessors which utilize a hypercube communication topology.

```
request(f)
{
    if (facility f is non-local) {
        send a request message to the site where f is
            located;
        wait for receipt of a successful
            acknowledgement. /* context switch */
    }
    else
        perform operations as in the sequential version;
}
release(f)
{
    if (facility f is non-local)
        send a release message to the site where f is
            located;
    else
        perform operations as in the sequential version
            except that an ACK is sent if the next process
            in queue is remote;
}
fac_server()
{
    for(;;) {
        wait for a message msg;
        switch(type of msg) {
        REQUEST:
            if (facility f is free) {
                flag f as occupied;
                send a successful ACK back;
            }
            else
                insert requesting process's pid at tail
                    of queue.
            break;
        RELEASE:
            if (waiting queue for facility f is not empty)
              {
                remove process pid from head of
                    queue;
                if process pid is local
                    insert reactivation record of pid
                        in calendar.
                else
                    send a successful ACK to pid.
            }
            else
                flag resource f as free;
        }
    }
}
```

Figure 7: Modifications to request( ) and release( ) using DSM approach

Stack Size (in bytes)	16	64	256	1024
Migration Cost	253	288	399	835

Table 1: Process Migration Cost (in microseconds)

Message Size (in bytes)	16	64	256	1024
Transmission Cost	215	242	353	793

Table 2: Message Transmission Cost (in microseconds)

We have conducted three experiments to evaluate the performance of the mobile process approach. First, we examined overheads related to process migration. Second, we used a simple tandem queue benchmark to compare the performance of mobile processes to DSM. Observe that both approaches give an analyst the same modelling view of the system, through a virtual shared memory mechanism. In a final experiment, we compared the performance of mobile processes to message-passing. As mentioned earlier, these two approaches give an analyst different modelling views of the system.

To preclude the possibility of a deadlock arising with a null message algorithm, we utilized service-time distributions that are shifted exponential.

4.1 Process Migration Latency

To evaluate the performance of the process migration primitive, we measured the time taken by a lightweight process to travel around a ring of processors. Table 1 shows the average time taken by a process to migrate from one processor to the next in the ring. Also included, for the purpose of comparison, are the corresponding overheads for message transmission (see Table 2). Using $t_{comm}(n)$ to denote total transmission time from one node to the next, and n to denote the message length (in bytes),

$$t_{comm}(n) = t_{startup} + t_{send} * n \qquad (1)$$

where $t_{startup}$ is the time to establish a connection, and t_{send} is the amount of time required for the transfer of a single byte of data. As can be derived from Equation 1 using the data given in Table 2, the startup time $t_{startup}$ is roughly 200 μ-seconds. With an efficient implementation of lightweight process migration, cost of migration will only grow linearly with stack size. The cost of this migration is only slightly higher (by about 40 μ-seconds) than the cost of message transmission.

4.2 Mobile Processes vs. DSM: Empirical Results

We used the tandem queue model described earlier as a simple benchmark, to compare the performance of mobile processes to DSM. Recall that both approaches give an analyst a uniform modelling view, where passive system components are modelled as shared objects. Therefore, a single program can be used to implement the benchmark, with the only modification required being the specification of a communication topology.

For a single remote request operation, we can evaluate these two approaches in terms of message transmission cost. As described earlier, a successful request operation in the DSM approach requires two-way messages, each containing the type of the message, a facility address, and the timestamp. Each such message requires precisely 12-bytes. Let t_{req_msg} and t_{rep_msg} denote the cost of send and reply messages, respectively. The cost of message transmission with DSM is given by

$$\begin{aligned} t_{DSM} &= t_{req\_msg} + t_{rep\_msg} \\ &= t_{comm}(12) + t_{comm}(12) \\ &= 2 * t_{startup} + 24 * t_{send} \quad \text{(from Equation 1)} \end{aligned}$$

We also measured the length of messages constituting migrated information, obtaining roughly 40 bytes per migration. The transmission cost with mobile processes works out to be

$$\begin{aligned} t_{PM} &= t_{migration\_info} \\ &= t_{comm}(40) \\ &= t_{startup} + 40 * t_{send} \end{aligned}$$

Because the value of $t_{startup}$ is 200 and t_{send} is less than 1, it is easy to see that

$$t_{DSM} > t_{PM}.$$

Furthermore, any performance difference between these two approaches is accentuated when locality is taken into account. With $m, (m > 1)$, consecutive accesses to the same remote site, the DSM approach requires a transmission cost of $m * t_{DSM}$. In the same situation, the mobile process approach would incur a cost of only t_{PM}. The latter cost is incurred solely by a single transmission, because after the first access request, all accesses are made locally.

Table 3 presents comparative timings for the benchmark run. Not surprisingly, the DSM approach performs poorly in comparison to the mobile process approach. In addition to the cost of the request operation and the non-local accesses, the bidirectional links and added communication

Processors	1	2	4	8
Process Mobility	249	118	59	31
Dist. Shared Mem.	249	325	423	567

Table 3: PM vs. DSM running the Tandem Queue Benchmark (in microseconds)

channels (for LP synchronization) reduce potential parallelism.

4.3 Mobile Processes vs. Message-Passing: Empirical Results

From the results shown in the previous subsection, we conclude that process migration is a very effective means for realizing a shared object (or say, mutual exclusion) mechanism. This is clear when migration is compared to distributed shared memory based techniques. In this subsection, we present results obtained by employing the same benchmark, in order to compare the two different views of realizing distributed simulations.

Figure 8 shows execution times (in seconds) obtained by executing two different programs (see Figure 2 and Figure 3 in Section 2) by scaling up the number of processors. With the support of lightweight process migration, it is not surprising to see that the Mobile Process approach performs almost as efficiently as the Message Passing approach. This is because both approaches generate the same number of message transfers.

With the Message Passing approach, each job arrival message requires the transfer of at least 12-bytes of information (including process id, message type, timestamp, etc.). With mobile processes, as described in the previous subsection, the arrival of a job corresponds to the migration of 40-bytes of information. The cost difference $(40 - 12) * t_{send}$ is not significant in comparison to message startup cost at time $t_{startup}$. Another reason for the slightly larger execution times with mobile processes is the overhead of process creation, unavoidable in open queueing network simulations.

In a final experiment, we increased the number of servers from 64 to 256 in the tandem queue model. The intention was to increase the load on each processor. As a consequence, granularity of computation is increased, resulting in better speedup for both approaches. This result is shown in Figure 9.

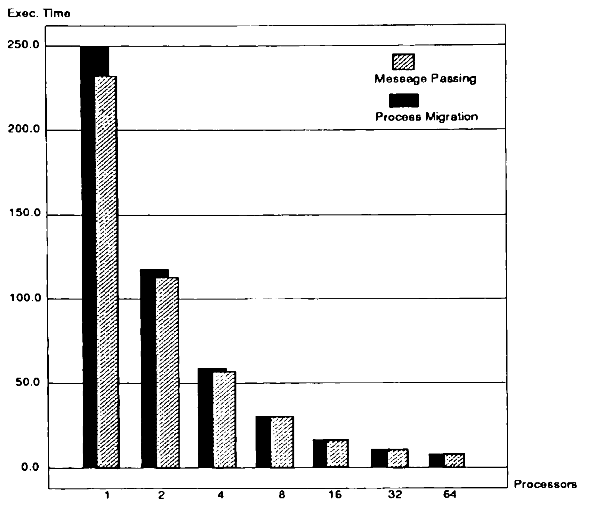

Figure 8: Process Mobility vs. Message-Passing

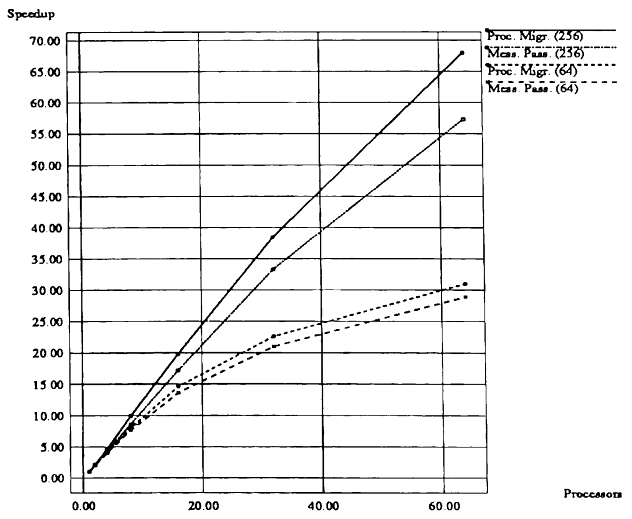

Figure 9: Speedup

5 CONCLUSIONS

The novel Mobile Process approach that we propose, for dealing with remote object access in distributed simulation, has proven quite successful. The advantages of one-time transmission, fixed communication topology and increased data locality make the approach more effective than methods based on distributed shared memory. Through mobile processes, the parallelization of process-oriented simulations on distributed-memory systems becomes feasible. Our experiments show that enhancing process-based models with a migration capability leads to simulations that are as efficient as the standard message-passing techniques in distributed simulation. Further, such models possess all the advantages of sequential process-oriented simulations.

ACKNOWLEDGEMENTS

This research was supported in part by NATO-CRG900108, NSF CCR-9102331, ONR-9310233, and ARO-93G0045.

REFERENCES

R. L. Bagrodia and W.-T. Liao (1990). Maisie: A language and optimizing environment for distributed simulation. In *Proceedings of SCS Multiconference on Distributed Simulation*, 1990.

G. Birtwistle, O. Dahl, B. Myrhaug, and K. Nygaard (1979). *Simula Begin*. Van Nostrand Reinhold, New York, 1979.

K. M. Chandy and J. Misra (1979). Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering*, 5(5):440–452, May 1979.

K. M. Chandy and J. Misra (1981). Asynchronous distributed simulation via a sequence of parallel computations. *Comm. ACM*, 24:198–206, April 1981.

J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield (1989). The Amber System: Parallel Programming on a Network of Multiprocessors. In *Symposium on Operating System Principles*, pages 147–158, 1989.

W. R. Franta (1977). *The Process View of Simulation*. North-Holland, Amsterdam, 1977.

D. W. Glazer and C. Tropper (1993). On Process Migration and Load Blancing in Time Warp. *IEEE Trans. on Parallel and Distributed Systems*, 4(3):318–327, 1993.

D. Jefferson (1985). Virtual time. *ACM Trans. on Programming Languages and Systems*, 7:404–425, July 1985.

D. Libes (1985). User-Level Shared Variables. In *Proceedings of the Summer USENIX Conference*, 1985.

J. Sang, K. Chung, and V. Rego (1993a). Computational schemes for efficient simulation of service disciplines. In *Proceedings of 26th Annual Simulation Symposium*, March 1993.

J. Sang, K. Chung, and V. Rego (1993b). Design and implementation of a simulation library using lightweight processes. In *Proceedings of the Summer USENIX Conference*, June 1993.

J. Sang, F. Knop, V. Rego, J. K. Lee, and C.-T. King (1993c). The Xthreads Library: Design, Implementation, and Applications. In *Proceedings of the COMPSAC*, 1993.

J. Sang and V. Rego (1993). Efficient Implementation of Threads Migration. Technical report, Computer Sciences Department, Purdue University, 1993.

H. D. Schwetman (1986). CSIM: A C-based process-oriented simulation language. In *Proceedings of the 1986 Winter Simulation Conference*, pages 387–396, 1986.

M. Stumm and S. Zhou (1990). Algorithms Implementing Distributed Shared Memory. *IEEE Computer*, 23:54–64, May 1990.

J. West and A. Mullarney (1988). ModSim: a language for distributed simulation. In *Proceedings of SCS Multiconference on Distributed Simulation*, 1988.