An Efficient Parallel Implementation of the Everglades Landscape Fire Model Using Checkpointing *

Fusen He and Jie Wu

Department of Computer Science and Engineering

Florida Atlantic University

Boca Raton, FL 33431

{fhe, jie}@cse.fau.edu

Abstract

This paper presents a low-communication overhead and high-performance data parallelism implementation of the Everglades Landscape Fire Model (ELFM) in a network of workstations (NOWs). ELFM is parallelized under Message Passing Interface (MPI). Checkpointing and rollback technologies are used to handle the spread of fire which is a dynamic and irregular component of the model. A parallel application model with the mixture of a variety of asynchronous and synchronous computation is developed. In this model, the asynchronous computation is dominant and synchronous computation is intermittent. The length of each synchronous computation also varies. Based on the developed model, a synchronous checkpointing mechanism is used in the parallel ELFM code under MPI. A simulation is conducted and results show that the performance of the ELFM under MPI is significantly enhanced by the application of checkpointing and rollback.

Key words: *Checkpointing, domain decomposition, Everglades Landscape Fire Model, Message Passing Interface, network of workstations, performance analysis, rollback, speedup.*

*This work was supported in part by a grant from South Florida Water Management District (SFWMD).

1 Introduction

With the advance of the network technology, network computing has entered into the main stream of solving scientific problems. Network computing is a process whereby a set of workstations connected by a network work collectively to solve a single large problem. As more and more organizations have already had high-speed networks/switches interconnecting many general-purpose workstations, the combined computational resources may exceed the power of a single high-performance computer. This trend has gained sufficient popularity to establish a new parallel processing paradigm: *Network of Workstations* (NOWs) [5]. A local area network (LAN) is a widely used network structure in a NOWs. Since LAN technology was not initially developed for parallel processing, communication overheads among workstations are still quite high [5]. This has placed severe constraints on obtaining high performance in a NOWs. The unacceptable performance of parallel Everglades Landscape Fire Model (ELFM) program using the network parallel programming environment *Express* is such an example [7].

The Everglades landscape is a vast freshwater marsh in South Florida and is one of the largest subtropical wetlands in the world. Fire has been an important ecological process in the Everglades and a primary factor shaping the Everglades vegetation patterns [9]. We cannot fully understand the Everglades without understanding the function of fire. Unfortunately, fire is a difficult process to experimentally manipulate, especially at a landscape level. This is because that the spread of fire is dynamic and probabilistic in nature. Computer simulation can reduce the time it takes to evaluate impacts and understand ecosystem dynamics. An Everglades Landscape Fire Model (ELFM) was developed to understand fire behavior in Water Conservation Area 2A (WCA 2A) in the Everglades. Figure 1 shows the geographical location of WCA 2A in the Everglades landscape.

Computer simulation can be applied to evaluate impacts and understand ecosystem dynamics. In order to speedup the simulation process, ELFM has been parallelized using Express [7] under several platforms such as UNIX workstations, CM-5 supercomputers, and Macintosh transputers. The parallel ELFM code has also been ported from Express to Message Passing Interface (MPI) [3]. The study in [1] shows that the major reason for the poor performance of the parallel ELFM code is the interprocessor communication overhead. It is also shown that the process synchronization consumes a huge portion of CPU time. In parallel ELFM simulation, when a fire occurs in landscape, it spreads. If a fire occurs near a boundary area of a subdomain simulated by a processor, it will spread to an adjacent subdomain that is simulated by a different processor. In this situation, data exchange is needed to simulate the process of fire spreading that acrosses the boundary of one subdomain to another. It is required that this data exchange be performed at the same simulation time step through process synchronization.

According to the fire behavior in landscape, the probability of fire occurrence is relatively small. Even when a fire occurs in a subdomain which is simulated by a processor, it may not be necessary to synchronize all the processors unless the fire spreads to other subdomains simulated by other processors. The main purpose of this study is to provide an efficient mechanism to support this type of parallel applications. Specifically, we try to enhance the performance of the parallel ELFM code, with MPI as its parallel programming environment, by using the checkpointing and rollback technique. The traditional checkpointing and rollback are generally used to address fault tolerance

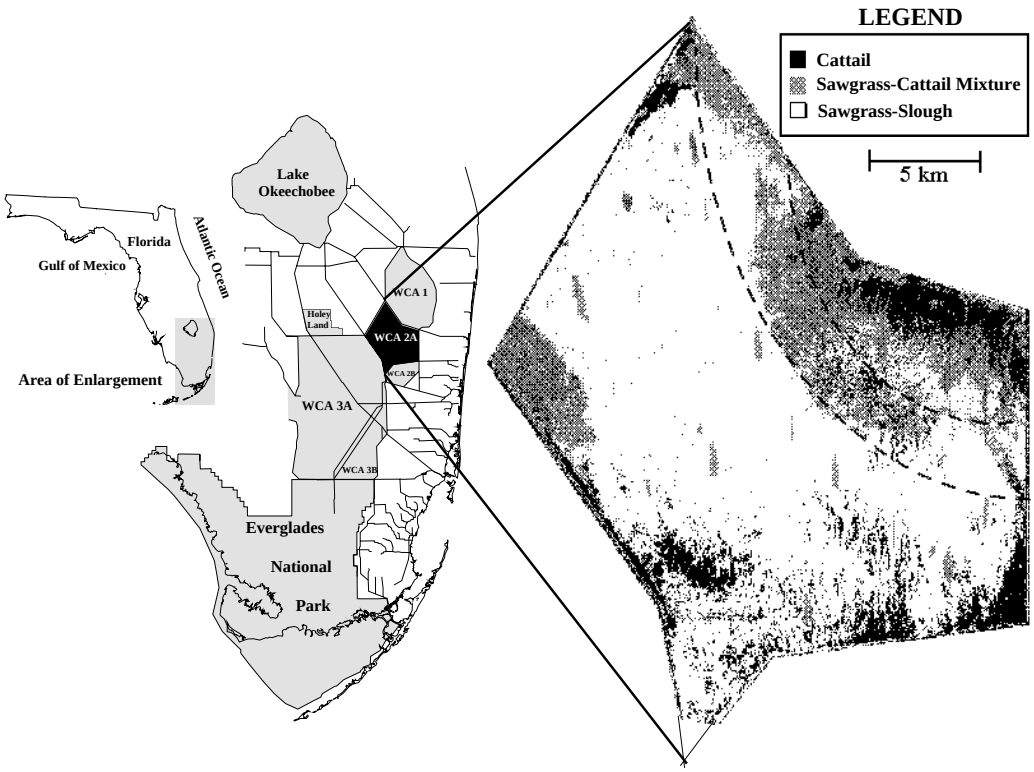


Figure 1: The geographical location of WCA 2A in the Everglades landscape.

issues [2]; however, we use them solely for the performance enhancement purpose in this study. The interval between two adjacent checkpoints (also called checkpoint interval) is adjustable. The heavy interprocessor communication can be reduced by a proper selection of the frequency of process synchronization among processors.

This paper is organized as follows: Section 2 discusses the current status of ELFM. Section 3 overviews several checkpointing and rollback techniques in NOWs. An approach which aims to reduce the heavy interprocessor communication overhead is discussed in Section 4. This approach is based on the checkpointing and rollback techniques. Section 5 presents the results of this study and shows on the improved performance of the parallel ELFM code using MPI. Section 6 concludes this paper.

2 The Current Status of ELFM

The ELFM code was used to simulate fire in WCA 2A of the north Everglades shown in Figure 1. The WCA 2A landscape, with area of 43,281 ha, is a mosaic of sawgrass marshes, sloughs, shrub

```
while(time in year not reaching the end of year) {
    while(time in day not reaching the end of day in a year) {
        while(time in hour not reaching the end of a day) {
            update fuel moisture hourly between rains;
            daily rainfall and lightning simulation;
            wind speed and directions simulation;
            if(fire ignites a cell) {
               change the simulation time step from hours to minutes;
               check adjacent cells to see if there is enough heat to ignite the fuel
                  and time needed to spread to the adjacent cells;
               while(fire is burning) {
                    check the adjacent cells to see if fire is going to
                       spread to the adjacent cells;
                    add newly ignited fires;
                    possible fire spotting simulation;
               }
            }
        }
    }
}
```

Figure 2: Algorithm 1: Fire spreading and spotting simulation in serial ELFM code.

and tree islands, and invasive cattail communities. The ELFM code simulates fire on a large spatial scale with a fine resolution of 20m $\times$ 20m which, in terms of grid cell, comes to 1755 $\times$ 1634. A basic assumption in the ELFM is that it is a spatial model with mostly nearest neighbor interactions except fire spotting, that is, a fire jumps from one area to another. Fire spreading is a special case in which a fire jumps (spreads) to its adjacent areas. Each cell is homogeneous, i.e., the same computation and communication structure is used. The model is designed as a parallel program with the ability to compile and run on UNIX workstations, the CM-5 supercomputers, and Mac Transputers without any change in code.

In the ELFM code, the time step of fire spreading and spotting simulation is in minutes and the fuel level is updated every hour. Process synchronization is performed on a daily base. Therefore, fire spreading and spotting simulation is computational intensive. The basic algorithm of the fire spread and spotting simulation in the serial ELFM code is shown in Figure 2

Because fire spreading and spotting is dynamic and probabilistic in nature, a good distribution (also called load balancing) of workload is difficult. The parallel ELFM code was initially ported directly from Express to MPI without any significant changes. However, the performance of the parallel ELFM code is unsatisfactory due to the interprocessor communication overhead [1]. The purpose of process synchronization is to make data consistent in simulation time space when data exchange between adjacent processors is needed. In Everglades landscape, the probability of fire occurrence is small. Even a fire occurs and spreads in the landscape, it usually affects a small part of the landscape rather than the entire landscape. If a fire does not spread to another subdomain which is simulated by another processor, there is no need to exchange data among processors in

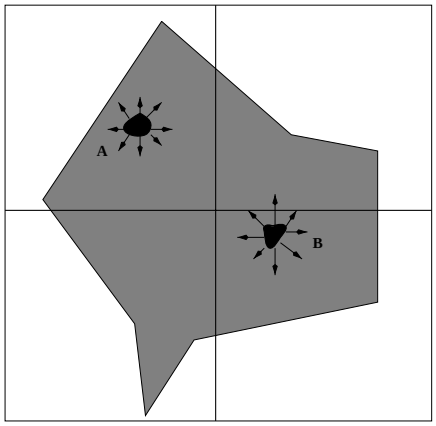


Figure 3: Examples of fire spread in landscape in ELFM.

this case. The fire spot A in Figure 3 shows such an example. The shadowed region is the effective computational domain and region is divided into four subdomains. Data exchange is only needed when a fire spreads across the boundary of a subdomain to another subdomain. In this case, fire spot A does not across to the neighbor subdomains. The fire spot B in Figure 3 shows an example in which a fire spot goes across to one of its neighbor subdomains.

The early version of the parallel implementation of the ELFM code uses a pessimistic approach. Process synchronization through collective communication is performed at each simulation step (either in minutes or in hour) even when there is no fire in the landscape. Since interprocessor communication overhead is still quite high in NOWs architecture, poor performance of the parallel ELFM code using this pessimistic approach can be expected. By analyzing the ELFM code, we have found that the occurrence of fire spreading is rare. We can use checkpointing combined with rollback techniques to enhance the performance of the parallel ELFM code. Data exchange is treated as message passing among processors in the specific NOWs. No message passing among processors are needed in regular simulation steps. Checkpoint (a set of local states) is made at a regular interval. Process rolling back to its checkpoint is needed when a fire spreads to its neighboring subdomains to keep simulation data consistent. This kind of approach is optimistic in nature.

One issue we need to address is process synchronization. In parallel/distributed computing, a *barrier* [10, 4] is used to perform this type of task. A barrier is a synchronization point in a parallel program at which all processes participating in the synchronization must arrive before any of them can proceed further. It is usually implemented as a function which may or may not take an argument. Once a process has called this function, it will not return until every other process has called it. Another type of process synchronization is one-sided communication. One-sided communication is extremely useful in the parallel ELFM code. Using one-sided communication, process synchronization is performed only when data exchange is needed and such synchronization is called conditional process synchronization. Unfortunately, the implementation of one-sided communication is still unavailable, we have to find another way to implement conditional process

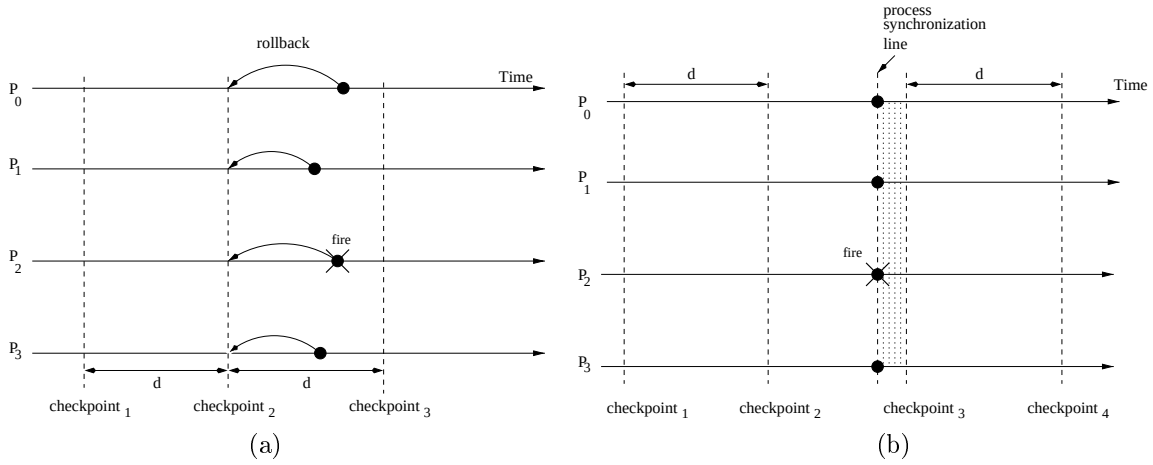


Figure 4: (a) Processes rollback in a 4-processor group. (b) Synchronize processes before message exchange.

synchronization. This issue will be discussed in Section 4.

3 Checkpointing and Rollback

In distributed systems such as NOWs, a global state is defined as a collection of local states, one from each processor in the NOWs. The checkpointing method [8, 6] is used to determine the global state. During the process execution, each processor periodically checkpoints its state by storing its execution state information into a stable storage such as a hard disk. Checkpointing is generally used in reliability study. In such an application, system states are stored regularly as checkpoints. When a failure causes an inconsistent state in the distributed system, it can rollback to a previous consistent state by simply restoring a prior checkpointing state. This rollback process is also known as rollback recovery. In ELFM, rollback recovery is needed when a global state becomes inconsistent, as in the case when a fire acrosses boundary of a subdomain, all the processors need to restore a previous state which is stored in the latest checkpoint.

A *strongly consistent set of checkpoints* consists of a set of local checkpoints such that no information flow takes place between any pair of processors during the interval spanned by the checkpoints. Checkpointing can be either synchronous, asynchronous, or a combination of both. Another choice is whether or not to log messages that a processor sends or receives. For parallel applications such as ELFM, synchronous checkpointing is the best choice since message exchange must be performed at the same physical process evolution time. Clearly, checkpoints produced by synchronous checkpointing form a strongly consistent set.

In the parallel ELFM code, we use checkpointing combined with rollback to enhance the performance of the program. To simplify our discussion, we consider an example of a NOWs consists of four workstations and the problem domain of the ELFM is partitioned into four subdomains

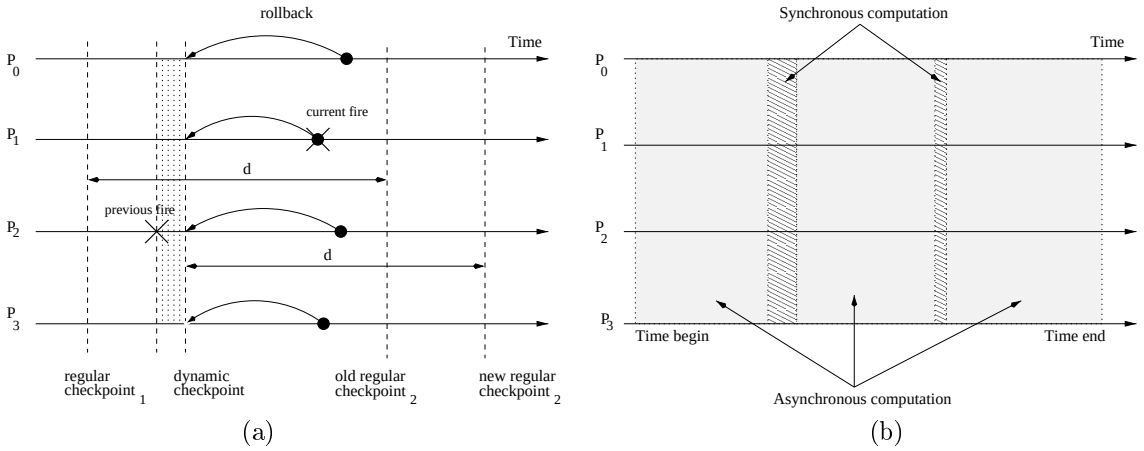


Figure 5: (a) Rollback with multiple message exchanges between a checkpoint interval. (b) Synchronous and asynchronous computations of an application in a NOWs with 4 workstations.

with each subdomain assigned to one distinct workstation. It can be easily extended to a generalized case with n workstations in a NOWs. Figure 4(a) shows a typical rollback process. The horizontal parallel lines represent the simulation time space (rather than the physical time space) in each processor. The vertical dotted lines represent synchronous checkpoints. d is the checkpoint interval, which is a constant in our simulation. The black dot on each horizontal line represents the simulation time step of the corresponding subdomain at the current physical time. Since each processor may have different workloads and different processing speed, if there is no process synchronization, the actual simulation time step at different processors may also be different. This means that processors run asynchronously. The cross sign ($\times$) in Figure 4(a) means that a fire occurs in processor P_2 and it is going to spread across the boundary of the subdomain (referred to as message exchange). All the processors rollback to their most recent checkpoints. After that, processors resume simulation from that checkpoint but still in the asynchronous mode. When reaching the time that message exchange is needed (the start of fire spreading and spotting simulation), all processors are synchronized and then perform message exchanges. This point is known as the synchronization point. Since a checkpoint is also a synchronization point, if a processor reaches a checkpoint while other processors are still behind this checkpoint, this processor is blocked for other processors to catch up. There exist several optimization methods, like lazy rollback (i.e. rolling back just the subdomains involved). However, they would not improve speedup, since it is based on the completion time of the last processor that finishes its simulation.

The shaded area in Figure 4(b) represents the period that the processors simulate fire spreading and spotting concurrently in the synchronous mode. After the completion of simulation on fire spreading and spotting, all the processors switch back to the asynchronous mode. The completion point of synchronous computation is logged as an new checkpoint. The checkpoint based on the checkpoint interval d is referred to as the *regular checkpoint*. Checkpoints 1, 2, and 4 in Figure 4(b) are regular checkpoints. The checkpoint immediately after the completion of synchronous computation is referred to as the *dynamic checkpoint*. Checkpoint 3 in Figure 4(b) is such an example.

Figure 5(a) shows the difference between regular and dynamic checkpoints. When multiple message exchanges are needed (because of multiple fires) in a regular checkpoint interval, all the processors rollback to their most recent dynamic checkpoints, restore their consistent states there, and resume simulation similar to those shown in Figures 4(a) and 4(b). If processors rollback to their most recent regular checkpoints, all the processors will enter into an infinite loop between the $regular\ checkpoint_1$ and the point of the current fire in Figure 5(a). By applying dynamic checkpointing, we avoid such infinite loops. Clearly, if there is no fire spreading and spotting during the simulation, only regular checkpoints are used. In the next section, we propose an algorithm based on the checkpointing and rollback mechanisms and show its application in parallelizing the ELFM code using MPI.

4 The Proposed Approach

This section introduces a low-communication overhead model based on checkpointing and rollback mechanisms. We begin with an analysis of the simulation time, discuss several relevant collective communication functions provided by MPI, and use checkpointing and rollback to parallelize the ELFM code.

Basic idea. The goal of developing a parallel version of a model is to allow a simulation to run in much less time than an equivalent serial version with the same numerical accuracy. By distributing workload over several processors, the amount of time taken to perform computation on an individual processor will be reduced. However, additional interprocessor communication and synchronization overheads make the program spend more time on simulation. Whether a parallel algorithm is successful or not depends on an appropriate balance between these two factors.

For parallel simulation in a NOWs, each workstation is assigned certain portion of the workload and works independently. We can name this kind of computation as *asynchronous computation*. However, when a neighbor interaction (such as fire spreading and spotting) occurs near the boundary of the subdomain simulated by a workstation, data exchange between workstations must be performed in order to make the result consistent. The corresponding workstations exchange data using the message passing mechanism, and data exchanges always occur at the same simulation time. Therefore, process synchronization is needed. This type of computation can be viewed as *synchronous computation*. The length of synchronous computation varies with time based on the duration of fire spreading and spotting. Figure 5(b) illustrates this type of application in a NOWs with four workstations.

When the computational load on each processor is low, if process synchronization is performed at every simulation step, it is obvious that interprocessor communication overhead will be large. The longer the checkpoint interval, the less the simulation time. However, if a fire spreads to adjacent subdomains simulated by other processors during the interval, the simulation time will increase. This is because the rollback process will force the system to return to an early state that has already been simulated. Therefore, more simulation time is needed. If we reduce the process synchronization interval, synchronization time will be wasted if there is no fire spreading and spotting to other subdomains at each checkpoint interval. The purpose of this study is to choose the checkpoint interval in order to gain a maximum possible speedup.

In order to keep consistent data, each processor needs to know the maximum number of simulation steps for each burning fire in the entire landscape, not just in the subdomain simulated by the local processor. MPI collective communication functions such as *MPI_Allgather* and *MPI_Allreduce* are used to collect the maximum number of simulation steps in the NOWs. Since the interprocessor communication in the current MPI implementation is sender/receiver based, the above mentioned collective communication functions synchronize the processors while collecting information. There is no need to use *MPI_Barrier*, a synchronization function in MPI, to perform the process synchronization.

The performance of a parallelized program can be referred to as speedup, which is the ratio of the computation time for a sequential computation to that of a parallelized version of the same computation. The theoretical speedup of a computation is proportional to the number of processors used in the computation. Since UNIX is a multiuser/multitask operating system, the elapsed physical execution time varies between individual runs. However, the CPU time dose not change. We use the CPU time to measure the performance of the parallel ELFM program.

Proposed approach. The ELFM is a program that involves a small amount of computation, but with huge amount of interprocessor communication if all the processes are synchronized at each simulation iterative step (in minutes). The most common approach for the parallelization of a spatial model like ELFM is *data parallelism*. This method decomposes the two-dimensional data domain of the ELFM into several subdomains, and each subdomain is stored on a separate processor. Each processor is only responsible for the specific subdomain that is assigned on it. A major issue arises in the data parallelism approach is how to maintain consistent data among the processors in the NOWs. The ELFM is a generic ecosystem "unit" model. Each element in the map needs the information on the adjacent elements to determine the state of the next iterative step during the process of simulation. Thus, unit elements lie on the borders of a processor must be able to communicate with unit elements on the borders of the adjacent processors at each iterative time step. In the present parallel implementation of the ELFM, each two-dimensional data domain is divided into n subdomains, and each subdomain is assigned to one processor. In order to handle the interprocessor communication among the adjacent processors, each subdomain is equipped with four additional cell edge strips with one from each of the four neighboring subdomains. Figure 6(a) shows the data configuration in the parallel ELFM. When a fire spreads across boundaries of subdomains simulated by processors in the NOWs, the adjacent processors exchange the adjacent cell edge strips by interprocessor communication to maintain consistent data set in the parallel ELFM.

The previous study [1] of the parallel ELFM code indicated that the synchronous computation is needed only when there are data exchanges between adjacent processors, which is needed only when a fire acrosses the boundary to another subdomain simulated by a different processor. Checkpointing is an ideal choice to improve the performance of the parallel ELFM code. Since data exchange among processors is performed at the same simulation time step, synchronous checkpointing will be the best choice. In our simulation, the synchronous checkpointing interval is measured by days.

The interprocessor communication in the current version of MPI is a two-sided communication. It is invoked at both sender and receiver sides. Regular send-receive communication requires matching operations by sender and receiver. This message-passing communication achieves two effects: communication of data from sender to receiver and synchronization of sender with receiver.

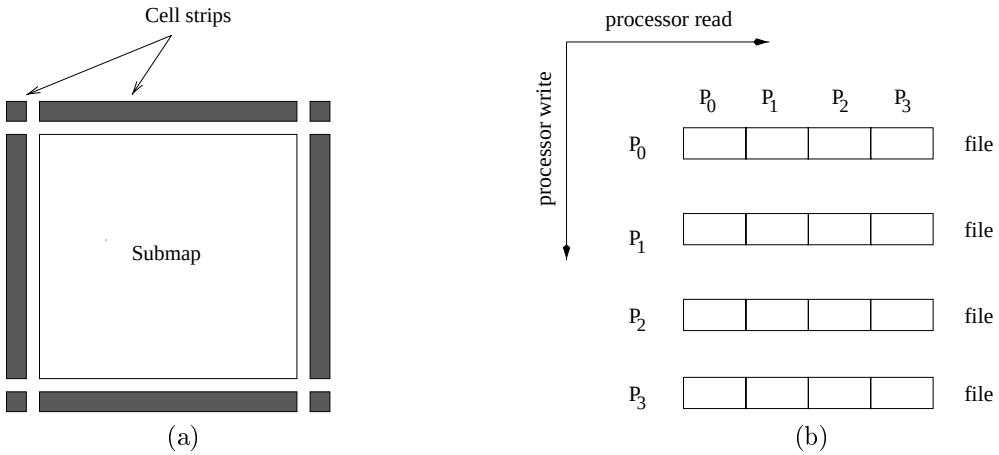


Figure 6: (a) Submap with edge cell strips. (b) Rollback information stored in files in a 4-processor NOWs.

However, in the parallel ELFM code, when a fire spreads across the boundary of a subdomain, only the processor that holds that subdomain has the information needs to be sent. This means that data to be transferred to other processors are available only on one side. It would be better if we could transfer data to other processors asynchronously. That is, sending data whenever it is ready and reading data when needed. The current MPI interprocessor communication functions always include send/receive pairs. Even the MPI nonblocking operations cannot meet our requirements. We have to use another way to realize this type of asynchronous one-sided interprocessor communication.

Sun Microsystems' Network File System (NFS) is a convenient choice. NFS is a remote file access mechanism defined in the UNIX operating system. NFS allows applications on one computer to access files on a remote computer as if it is a local file. In the parallel ELFM code, data need to send out can be stored into files in a hard disk. Processors read these files when needed. By doing so, unnecessary interprocessor communications can be avoided, and therefore, it provides an effective means to implement process synchronization.

During the process of simulation, each processor keeps a set of flags which are referred to as rollback flags. This flag set stores the status information of all the processors in a NOWs. Each flag set is stored as a data file in the hard disk and the size of the flag set is equal to the number of processors in the NOWs. These files are referred to as the rollback files. The number of files is also equal to the number of processors. The position of a rollback flag for a specific processor in the file matches the processor id of that processor. Reading and writing operations on files are performed based on rules described in Figure 6(b): Each processor reads the complete rollback flag set from the file assigned to it. However, processor P_i only updates rollback flags which store the rollback information of this particular processor. That is, the ith position of all the data files in Figure 6(b). This kind of operations can be expressed as "reads in row and writes in column". The rollback flag set is checked by a processor on a daily base.

```
while(time in year not reaching the end of year) {
    while(time in day not reaching the end of day in a year) {
        if(time in day reaches checkpoint) {
            log the day of checkpoint as idayck;
        }
        while(time in hour not reaching the end of a day) {
            update fuel moisture hourly between rains;
            daily rainfall and lightning simulation;
            wind speed and directions simulation;
            if(fire ignites a cell) {
                change the simulation time step from hours to minutes;
                check adjacent cells to see if there is enough heat to ignite the fuel
                    and time needed to spread to the adjacent cells;
                if(fire spread across boundary) {
                    set rollback control flag to y and save it to file;
                    log the current day as idayst and save it to a file;
                    set data exchange flag to true;
                    rollback to the previous checkpoint;
                    restore global state at checkpoint;
                }
                else {
                    after current day simulation, each processor reads
                        rollback information from files;
                    if(rollback flag is set to y) {
                        read the minimum idayst from starting time files;
                        rollback to the most recent checkpoint and restore states; }
                    if(time in day reaching the day of checkpoint) {
                        store a copy of state as a checkpoint;
                        log the regular checkpoint as idayck; }
                }
```

Figure 7: Algorithm 2: Parallel ELFM code with checkpointing and rollback.

Just before a fire spreads across the boundary to another subdomain simulated by a different processor, the processor executing the current simulation sets its rollback flag to true and updates the data files which store the rollback flag set. This processor also creates a starting time file that stores the time at which the fire begins to spread across the boundary to other subdomains simulated by other processors. Then this processor rollbacks to its most recent checkpoint. It restores the saved state of that processor at the checkpoint and resumes simulation from the checkpoint in the asynchronous mode. However, it switches to the synchronous mode once it reaches the starting time.

The operations for those processors which do not initiate the rollback process are described as follows: These processors read the rollback flags from the rollback flag files. If they find that some of these flags are set to true, these processors reset them back to false. They also select the minimum starting time from the corresponding starting time files. These processors then rollback

```
while(fire is burning) {
    if(process synchronization flag is set)
      synchronize processes each time step in minutes;
      if(data exchange flag is set and time in day is idayst)
        exchange data on the boundary of subdomain;
        check the adjacent cells to see if fire is going to
            spread to the adjacent cells;
            add newly ignited fires into consideration;
            possible fire spotting simulation;
            update simulation time in minutes;
        }
      }
      if(synchronous computation completed) log current day as idaybk;
    }
    if(process synchronization flag is set) synchronize processes each day;
    if(time in day reaches idayck or idaybk) log the global states in main memory;
    }
}
```

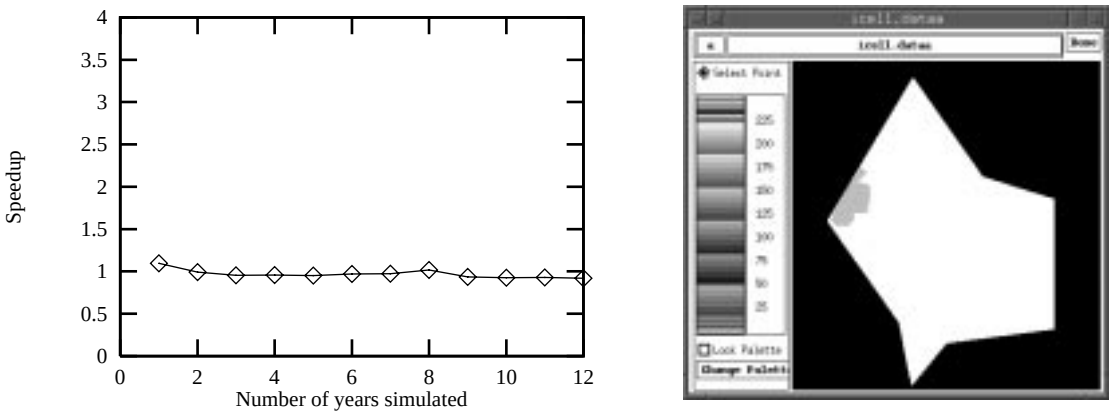
Figure 8: Algorithm 2: Parallel ELFM code with checkpointing and rollback (continued).

to their most recent checkpoints, restore their states at the checkpoints, and resume the simulation in the asynchronous mode. However, these processors will switch to the synchronous mode once their simulation time reaches the minimum starting time they read from starting time files. All processors will change back to the asynchronous mode once the current fire stops. The mechanism that resets rollback flags back to false avoids the infinite loop that may occur in the parallel ELFM. If the flag is not set to false, after the synchronous computation, the processors read the rollback flag set again and get an incorrect conclusion that message exchange is needed. In order to keep the stored data up-to-date, the *fsync* function in UNIX should be called each time when data writing is performed. *fsync* forces the UNIX operating system to flush data in memory buffer to a hard disk.

The algorithm for fire spreading and spotting simulation used in the parallel ELFM code with checkpointing and rollback mechanisms is shown in Figures 7 and 8, where regular checkpoint is represented by *idayck* and the dynamic checkpoint is represented by *idaybk*. *idayst* is the starting time where processors enter into synchronous mode. In the proposed approach, the states of the most recent checkpoint are stored in the processor's main memory. The size of the data is $3 \times 1755 \times 1634/n$, where n is the number of processors in the NOWs.

5 Results and Discussion

The parallel ELFM using the proposed approach is implemented using MPICH, which is an MPI implementation provided by Argonne National Laboratory. The computing environment is a Sun Sparc V workstation NOWs running Solaris. These workstations are interconnected by a 10 Mbits



(a) (b)

Figure 9: (a) Speedup of the parallel ELFM using MPI without checkpointing. (b) Landscape patterns of the WCA 2A in the Everglades after a 1-year period simulation by the parallel ELFM code.

Ethernet.

We use speedup to measure the computational performance of the parallel ELFM using MPI. In order to show the improvements achieved by the proposed approach, we first look at the speedup of the parallel ELFM using Express. The performance analysis in [7] indicated that the four processor version of the parallel ELFM was slower than the one processor code by a factor of four; the four processor version took roughly 10 minutes to simulate one day, and the one processor version clocked in at about 2.6 minutes. There is a light variation in these values between individual runs of these models, however, due to network traffic and other factors. The true serial version of the code runs at a rate of roughly 11 years simulation in 90 minutes, or 0.02 minutes per day. Thus the performance of the parallel ELFM code using Express is unacceptable.

In an early study [1], the parallel ELFM code using MPI has been run on a NOWs with four Sun Sparc V workstations. Figure 9(a) shows the speedup of the parallel ELFM using MPI without the checkpointing technique. The simulated simulation times are from 1-year to 12-year periods. This version of the parallel ELFM code uses a pessimistic approach. That is, process synchronization is conducted at every simulation time step. The serial ELFM code also runs on each individual workstation in the NOWs. Compared to the results using Express [7], the computational performance of the parallel ELFM code is improved; however, it is still unsatisfactory.

In the present study, the serial and parallel executions are both run on the four workstations. All the results are based on a 1-year period simulation. First, we executed the parallel ELFM code without checkpointing and rollback, the program execution time on each of the processor in the 4-workstation NOWs are shown in Table 1.

Since the workstations are usually used as multitask and multi-user systems, the workload varies from processor to processor and the elapsed time of the each parallel execution also varies with different workloads. In order to analysis the computational performance of the parallel ELFM, we

Table 1: Parallel program execution time in a 4-workstation NOWs

Process Id	Real Time	User Time	System Time
0	4447.61	837.92	920.38
1	4447.52	580.34	460.66
2	4447.49	767.69	941.38
3	4447.38	605.42	478.52

Table 2: Parallel program execution CPU time in a 4-workstation NOWs using checkpointing but without rollback

Interval	Average	Longest	Shortest	Interval	Average	Longest	Shortest
5	509.95	603.04	438.25	10	401.44	469.38	358.81
15	380.84	440.97	342.43	20	353.35	404.46	319.43
25	341.12	389.31	303.34	30	337.14	385.24	298.21
35	329.64	374.52	289.73	40	327.52	370.00	287.55
45	323.90	367.11	280.89	50	321.56	362.84	277.39
55	318.25	361.86	275.11	60	319.49	359.57	274.94

focus on CPU time, rather than the elapsed time. A processor's CPU time is composed of two parts. One is known as *user time*, and the other is *system time*. User time is the CPU time used while executing instructions in the user space of the calling process, and system time is the CPU time used by the system on behalf of the calling process.

The computational performance of the parallel ELFM without using checkpointing and rollback [1] indicated that most of the numerical computation is related to the user time, almost all the system time and part of the user time are related to interprocessor communication. The processor idle time is the real elapsed time minus user time and the system time. This is the time that processors wait for the operating system to process jobs submitted by other users. Since there are no dedicated workstations can be used to parallel computing in our computing environment, the idle time on each processor is much larger than the user time and the system time.

To study the influence of the checkpoint interval and rollback to the computational performance of the parallel ELFM code, we first perform a simulation of the parallel ELFM using only checkpointing but without rollback. In this model, processors only synchronize at a certain given checkpoints. The parallel ELFM with checkpointing only synchronize processors at each checkpoint. This is the ideal case of our checkpointing and rollback algorithm. However, if a fire spreads to the adjacent subdomains simulated by other processors in the checkpoint interval, the result will be inaccurate. The numerical accuracy can be enhanced by reducing the checkpoint interval, but it can never reach the level as the one with a rollback process. The program execution times of this parallel ELFM code for different checkpoint intervals are shown in Table 2.

The application of checkpointing and rollback in the parallel ELFM significantly reduces the

Table 3: Parallel program execution times in a 4-workstation NOWs.

(a) Processor 0

Checkpoint Interval	Real Time	User Time	System Time
20	1019.61	401.10	92.78
40	791.60	431.52	55.66
60	822.01	320.73	44.52
80	662.88	387.18	26.83
120	779.84	497.70	27.15

(b) Processor 1

Checkpoint Interval	Real Time	User Time	System Time
20	1004.98	380.04	69.46
40	776.16	394.44	38.29
60	808.17	414.05	32.61
80	649.54	355.56	19.53
120	766.03	414.20	19.80

(c) Processor 2

Checkpoint Interval	Real Time	User Time	System Time
20	1007.11	321.68	100.27
40	779.25	320.73	54.54
60	810.36	380.16	46.16
80	651.52	304.88	26.59
120	768.15	354.65	26.64

(d) Processor 3

Checkpoint Interval	Real Time	User Time	System Time
20	1009.27	393.01	66.98
40	781.29	405.51	38.08
60	812.71	446.09	32.72
80	653.56	396.22	19.30
120	770.45	443.09	19.36

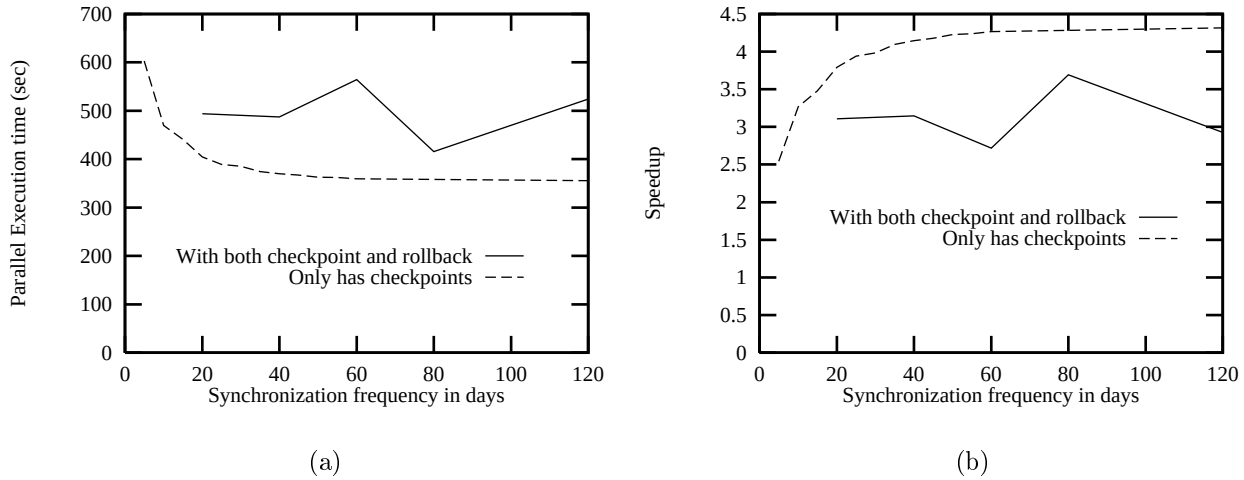


Figure 10: (a) Program execution CPU time of the parallel ELFM code vs. synchronization frequency in a 4-workstation NOWs. (b) Speedup of the parallel ELFM code vs. synchronization frequency in a 4-workstation NOWs.

interprocessor communication overhead of the parallel ELFM program. The program execution times of the parallel ELFM code with checkpointing and rollback running in a 4-workstation NOWs with different checkpoint interval are shown in Table 3.

Compared with the execution time of the parallel ELFM without using the checkpointing mechanism, the program execution time is significantly reduced. Figure 10(a) compares the program execution CPU time of the parallel ELFM program with only checkpointing to that with checkpointing and rollback techniques.

Figure 10(b) shows the comparison in terms of speedup. A superlinear speedup is obtained for execution only with process synchronization. Compared with the serial ELFM code, the parallel ELFM code uses only a quarter of the memory that the serial version uses. This may be the reason for this superlinear speedup. We can see that the program execution CPU time with checkpointing and rollback takes a little longer than the one with only checkpointing (process synchronization). This is because that rollback process takes some extra time. Since the probability of fire spreading in landscape is small, the probability of a rollback process to be invoked is also small. When there is no fire spreading and spotting during the process of simulation, the parallel ELFM with checkpointing and rollback reduces to the parallel ELFM with only checkpointing. When the checkpoint interval varies from 20 to 120 days, the speedup of the parallel ELFM program fluctuates in the range of 2.6 to 3.7. The average speedup is above 3. The computational performance of the parallel ELFM code is significantly enhanced with the checkpointing and rollback techniques. Figure 9(b) shows the landscape patterns after a 1-year period simulation. The grey area in the landscape indicates that fires occurred at that area.

6 Conclusion

We have reported a study of parallelization of Everglades Landscape Fire Model (ELFM) using Message Passing Interface (MPI). The ELFM code has been successfully ported to MPI. We have studied the checkpointing and rollback techniques and have applied the synchronous checkpointing mechanism combined with the rollback technique to parallelize the ELFM code using MPI. The performance analysis shows that a better speedup is obtained compared to the parallel ELFM code without the checkpointing and rollback techniques. This study indicates that for certain type of parallel applications such as ELFM, if the probability of interprocessor communication is small, the checkpointing and rollback techniques are useful to reduce the simulation time.

The future work will focus on generalization of the parallel computation model with the mixture of a variety of asynchronous and synchronous computations. Parameters which affect the performance of the parallel applications, such as the synchronization cost, the asynchronous and synchronous computation ratio, load balancing, etc., will be studied both theoretically and numerically.

References

[1] He F., J. Wu, C. Fitz, F. Sklar, and Y. Wu. A Parallel Implementation of the Everglades Landscape Fire Model Using Message Passing Interface. Report to South Florida Water Management Disctrict, Florida Atlantic University, March 1998.

[2] Jalote P.. *Fault Tolerance in Distributed Systems.* Prentice Hall, Inc., Englewood Cliffs, NJ, 1994.

[3] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, version 1.1. Available via anonymous ftp from *ftp.mcs.anl.gov*, June 1995.

[4] Pacheco P. S. *Parallel Programming with MPI.* Morgan Kaufmann Publishers, Inc., San Francisco, California, 1997.

[5] Panda D. K. , and L. M. Ni. Special Issue on Workstation Clusters and Network-Based Computing. *Journal of Parallel and Distributed Computing*, **40**:1 – 3, 1997.

[6] Wang Y.-M., Y. Huang, K.-P. Vo, P. Y. Chuang, and C. Kintala. Checkpointing and Its Applications. In *Proceedings of the 25th Int'l Symp. on Fault-Tolerant Computing*, pages 22–30, Pasadena, CA, 1995.

[7] Wille L. T., and P. J. Ulintz. Parallel Simulations of Fire in the Everglades – Performance Analysis of Algorithm. Report to South Florida Water Management District, Florida Atlantic University, December 1996.

[8] Wu J. *Distributed System Design.* CRC Press, Boca Raton, FL, 1998.

[9] Wu Y., F. H. Sklar, K. Gopu, and K. Rutchey. Fire Simulations in the Everglades Landscape Using Parallel Programming. *Ecological Modelling*, **93**:113–124, 1996.

[10] Yang J.-S. and C.-T. King. Designing Tree-Based Barrier Synchronization on 2D Mesh Networks. *IEEE Transactions on Parallel and Distributed Systems*, **9**(6):526–534, June 1998.