A Parallel Implementation of the Everglades Landscape Fire Model in Networks of Workstations *

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). Checkpointing and rollback techniques were used to handle the spread of fire which is a dynamic and irregular component of the model. A synchronous checkpointing mechanism was used in the parallel ELFM code using Message Passing Interface (MPI). The speedup and performance of the parallel program were also studied. Results show that the performance of ELFM using MPI is significantly enhanced by using the checkpointing and rollback mechanisms.

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) [1].

A NOWs can be organized as a "cooperative cluster" to perform parallel/distributed computation for a single application. Each individual workstation can be assigned a part of a given problem and these parts can be computed concurrently between synchronization points. When the computation reaches these points, the participating workstations pause in their computation stage and enter a communication stage. During the communication stage, these workstations exchange messages containing the intermediate results needed in the

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

A. Sivasubramaniam, M. Lauria (Eds.): CANPC'99, LNCS 1602, pp. 1–15, 1999.

next computation stage. 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 [1]. This has placed severe constraints on obtaining high performance in a NOWs. The unacceptable performance of the parallel implementation of the Everglades Landscape Fire Model (ELFM) using the network programming environment *Express* is such an example [3].

The Everglades landscape is a vast freshwater marsh in South Florida and is one of the largest subtropical wetlands in the world. The Everglades has changed dramatically during this century with vast areas being converted to urban and farming land use. These changes may significantly affect efforts to restore natural vegetation and hydroperiods in the remaining Everglades. Fire has been an important ecological process in the Everglades and a primary factor shaping the Everglades vegetation patterns. 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. Recently, an Everglades Landscape Fire Model (ELFM) [8] was developed to understand fire behavior in Water Conservation Area 2A (WCA 2A) in the Everglades.

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 [3] in 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) [4]. The study in [2] showed that the major reason for the poor performance of the parallel ELFM code is the heavy 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 subdomain. 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 techniques. The traditional checkpointing and rollback are normally used to address fault tolerance issues [5]; 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 a NOWs. An approach aiming to reduce the heavy interprocessor communication and synchronization overhead is discussed in Section 4. Section 5 presents the results of this study and shows the improved performance of the parallel ELFM code using MPI. Section 6 concludes this paper.

2 Everglades Landscape Fire Model (ELFM)

The ELFM code was used to simulate fire in the Water Conservation Area 2A (WCA 2A) in the northern Everglades. The WCA 2A landscape, with an area of 43,281 ha, is a mosaic of sawgrass marshes, sloughs, shrub 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. ELFM is a spatial model with mostly nearest neighbor interactions except *fire spotting* in which a fire jumps from one area to another. *Fire spreading* is a special case in which a fire jumps (spreads) to its adjacent areas only. We assume that each cell in the landscape is homogeneous, i.e., the same computation and communication structure is used. The ELFM code is portable with its ability to compile and run on UNIX workstations, CM-5 supercomputers, and Macintosh Transputers without any significant changes in code.

In the current ELFM code, the simulation time step of fire spreading and spotting is measured in minutes and the fuel level (a static component in the fire model) is updated every hour. Process synchronization is performed on a daily base. Therefore, the simulation on fire spreading and spotting is computational intensive.

The early version of the parallel implementation of the ELFM code uses a pessimistic approach. Process synchronization through interprocessor communication is performed at each simulation step (either in minutes or in hours) even when there is no fire in the landscape. Since interprocessor communication overhead is still quite high in a NOWs, this pessimistic approach results in a poor performance of the parallel ELFM code [2]. By analyzing the ELFM code, we have found that the occurrence of fire spreading and spotting is rare. Even a fire occurs and spreads in the landscape, it usually affects a small portion of the landscape rather than the entire one. If a fire does not spread to another subdomain simulated by another processor, there is no need to exchange data among processors. We can use *checkpointing* (saving a set of local states) combined with *rollback* (processes rolling back to their checkpoints) to enhance the performance. In this approach, data exchange is treated as message passing among processors in a NOWs. No message passing among processors is needed in regular simulation steps. Checkpointing is made at a regular interval. Roll-

back is needed only when a fire spreads to its neighboring subdomains to keep simulation data consistent.

3 Checkpointing and Rollback

For parallel processing in a NOWs, a global state is defined as a collection of local states, one from each workstation in the NOWs. In the ELFM, the state is a set of numerical data which determines the evolution of the ecosystem in the Everglades. The checkpointing method [6], [7] is usually used to save the global state. During the normal execution, each processor periodically checkpoints its state by storing its execution state into a stable storage such as a hard disk. Checkpointing is normally used to achieve fault tolerance. In such an application, system states are stored regularly as checkpoints. When a failure causes an inconsistent state, it can rollback to a previous consistent state by simply restoring a prior checkpointing state. This rollback process is also known as rollback recovery.

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 the 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.

During the simulation, 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. This process is referred to as rollback.

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 with each subdomain assigned to a distinct workstation. It can be easily extended to a generalized case with n workstations in a NOWs. We refer to each workstation as a processor. Figure 1 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 dashed 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 $\times$ sign in Figure 1 means that a fire occurs in processor P_2 and it is going to spread across the boundary of the subdomain (re-

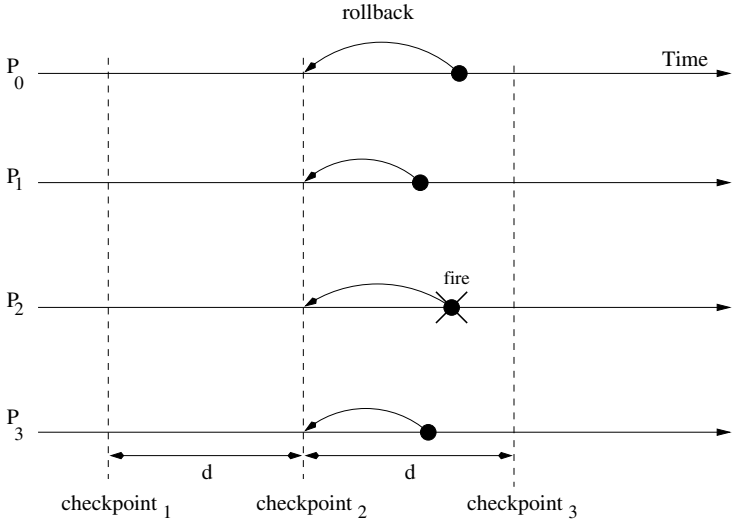

Fig. 1. Rollback process in a 4-workstation NOWs.

ferred 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 in our case, since it is based on the completion time of the last processor that finishes its simulation.

The shaded area in Figure 2 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. A checkpoint based on the checkpoint interval d is referred to as a *regular checkpoint*. Checkpoints 1, 2, and 4 in Figure 2 are regular checkpoints. A checkpoint immediately after the completion of synchronous computation is referred to as a *dynamic checkpoint*. Checkpoint 3 in Figure 2 is such an example.

Figure 3 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 dy-

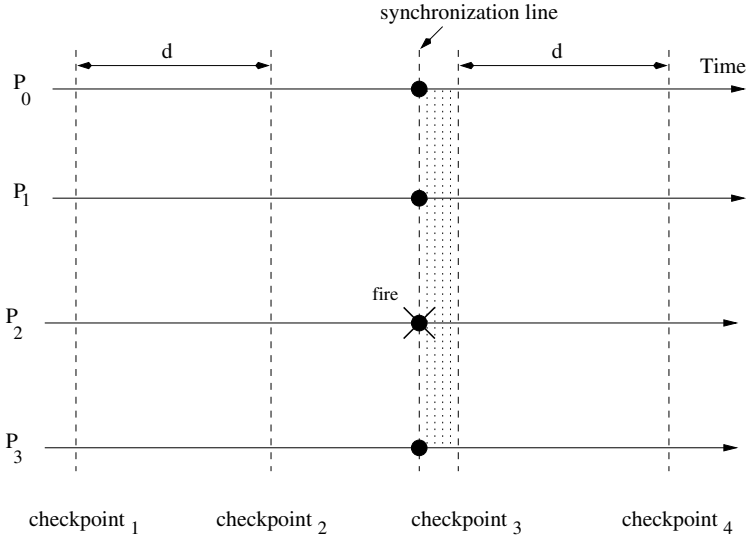

Fig. 2. Synchronize processors before message exchange.

namic checkpoints, restore their consistent states there, and resume simulation similar to those shown in Figures 1 and 2. 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 3. 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 start with a mathematical model for the estimation of simulation time, discuss several relevant collective communication functions provided by MPI, and use checkpointing and rollback to parallelize the ELFM code.

4.1 Mathematical model

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

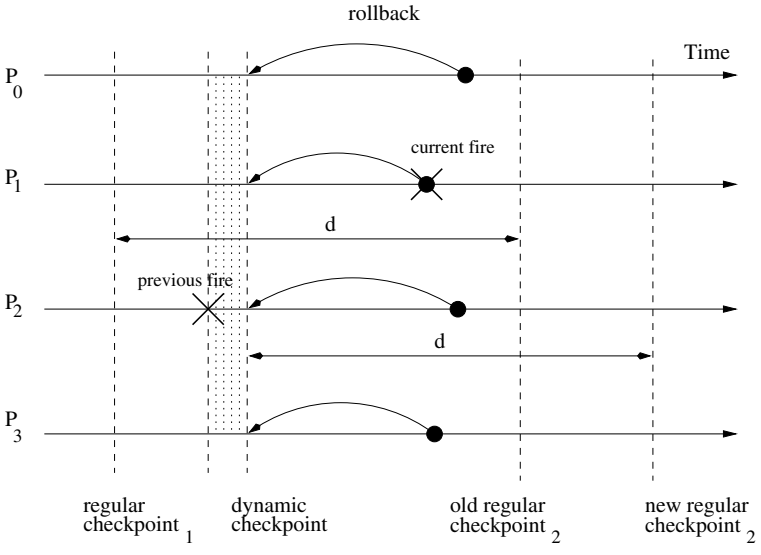

Fig. 3. Rollback with multiple message exchanges between a checkpoint interval.

of time taken to perform computation on an individual processor should 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 a balance between these two factors.

For parallel simulation in a NOWs, each workstation is assigned part 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 4 illustrates this type of application in a NOWs with four workstations.

Suppose that the probability of message passing among processors is p, the cost for message passing is c, the cost for process synchronization is s, the process synchronization interval (also called checkpoint interval) is d, the number of steps needed for the simulation is N, the total workload of the parallel program is W, the number of processors in the NOWs is n, and the processor processing speed is v_p which is the amount of workload the processor can process per unit time,

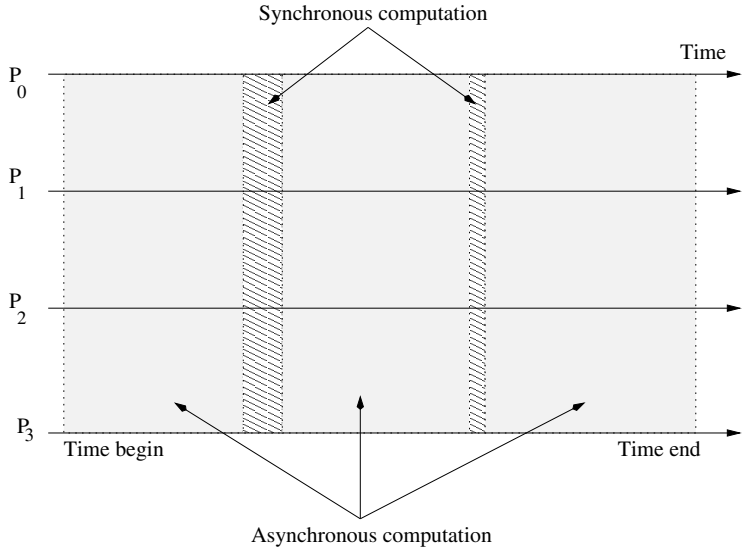

Fig. 4. Synchronous and asynchronous computations of an application in a NOWs with four workstations.

then the execution time of parallel program can generally be expressed as,

$$T = T_e + T_c + T_s + T_r = max\{\frac{W}{n \times v_p}\} + p \times N \times c + \frac{N \times s}{d} + T_r(p, d, v_p, N)$$

here, T_e is the execution time for effective workload, T_c is the execution time for interprocessor communication, T_s is the execution time for process synchronization, and T_r is the execution time for rollback and is a function of p, d, v_p, and N. If the workload is uniformly distributed, then $W = Nw$. w is the workload for each simulation step. $\frac{w}{n}$ is the workload on each processor per simulation step. Normally, c and s are much larger than $\frac{w}{n}$. When $\frac{w}{n}$ is small, it is obvious that if process synchronization is performed at every simulation step, that is $d = 1$, the 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 paper is to study how to choose the checkpoint interval to gain a maximum possible speedup.

In the parallel ELFM, 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 ideal speedup of a computation is proportional to the number of processors used in the computation.

Since UNIX is a multiuser/multitask operating system, the 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. The speedup of the parallel ELFM program can be expressed as follows,

$$Speedup = \frac{Average\ sequential\ CPU\ time}{Average\ parallel\ CPU\ time}$$

4.2 Application of checkpointing and rollback in parallel ELFM

The previous study [2] of the parallel ELFM code indicated that the synchronous computation is needed only when there are data exchanges between adjacent processors. This occurs when a fire acrosses the boundary to another subdomain simulated by a different processor. Checkpointing with rollback 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 goals: communication of data from sender to receiver and synchronization of sender with receiver. However, in the parallel ELFM code, when a fire spreads across the boundary of a subdomain, only the processor 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. The receiving processors do not know in advance when the relevant information will be sent to them. Regular send-receive commands cannot be placed in respective sending processors and receiving processors. It would be better if we can transfer data to receiving processors asynchronously. That is, sending data whenever it is ready at sending processors and reading data when needed at receiving processors. Even the MPI nonblocking operations cannot meet our requirements. We have to use another way to achieve 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 system to access files on a remote system as if it is a local file. In the parallel ELFM code, data need to be sent 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 that 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 5: 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 5. 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.

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 that 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, i.e., the start of a fire crossing the boundary.

The operations for those processors which do not initiate the rollback process are described as follows: 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 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 switch 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

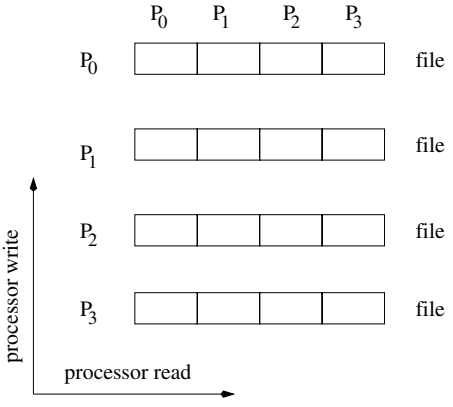

Fig. 5. File operations on rollback information in a 4-workstation NOWs.

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.

In the proposed approach, the most recent checkpoint of each processor is stored in the main memory of each processor. 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 set of Sun Sparc V workstations running Solaris. These workstations are interconnected by a 10 Mbits Ethernet.

We use speedup to measure the 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 [3]. The performance analysis in [3] indicated that the four-processor-version of the parallel ELFM was slower than the one processor code by a factor of about 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 [2], the parallelized ELFM code using MPI has been run on a NOWs with four workstations. Figure 6 shows the speedup of the parallel ELFM using MPI without using the checkpointing technique. This version of the parallel ELFM code uses a pessimistic approach. That is, processor synchronization is conducted at every simulation time step. The sequential version of the ELFM

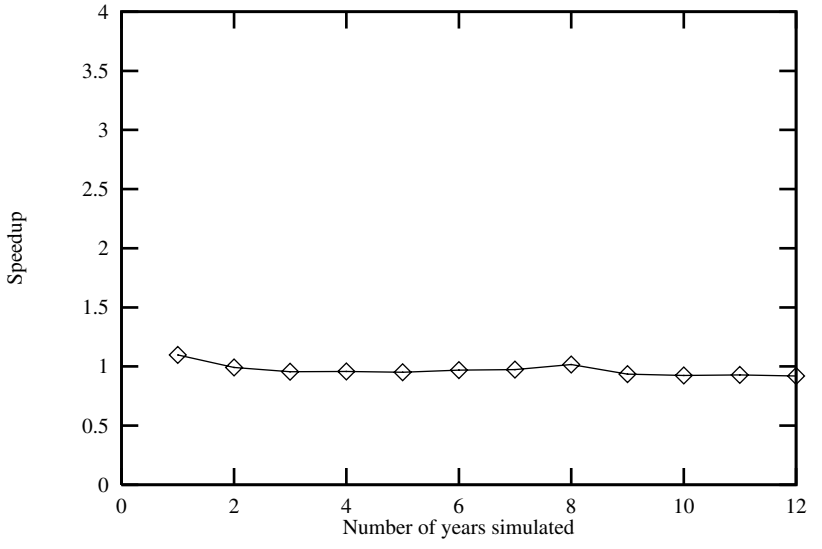

Fig. 6. Speedup of the parallel ELFM using MPI by a pessimistic approach.

code also runs on each individual workstation in the NOWs. Compared to the results using Express, the performance of the parallel ELFM code is improved; however, it is still unsatisfactory.

Since workstations are usually used as a multitask and multi-user system, the workload varies from processor to processor and the execution time also varies with different workloads. In order to analyze the performance of the parallel ELFM, we focus on CPU time, rather than elapsed time. A process'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. Most of computational costs are reflected in the user time, almost the entire system time and part of user time are related to interprocessor communication. The processor idle time is the actual elapsed execution time less the user time and the system time. The idle time on each processor is much larger than the user time and the system time.

In order to study the influence of the proposed algorithm on the performance of the parallel ELFM, we first performed a simulation of the parallel ELFM using checkpointing, but without rollback. In this model, processors only synchronize at certain given checkpoints. The parallel ELFM with only checkpointing synchronizes 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 NOWs, 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.

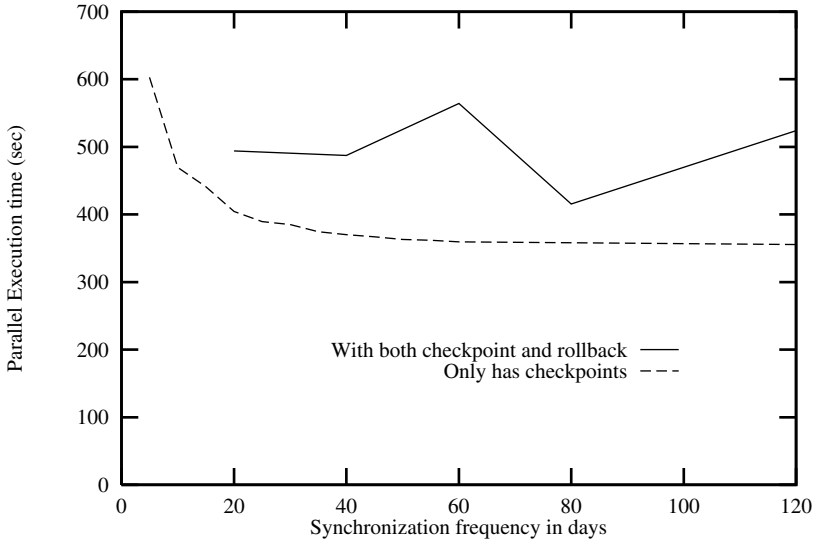

Fig. 7. Program execution time of the parallel ELFM vs. synchronization frequency in a 4-workstation NOWs.

The application of checkpointing and rollback techniques in the parallel ELFM significantly reduces the interprocessor communication overhead of the parallel ELFM program. Compared with the execution time of the parallel ELFM without using checkpointing mechanism, the system time is greatly reduced. Figure 7 compares the execution time of the parallel ELFM program with only checkpointing to that with checkpointing and rollback.

Figure 8 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 might be the reason for this superlinear speedup. We can see that the execution time with checkpointing and rollback takes a little longer than the one with only process synchronization. This is because the rollback process takes some extra time. Since the probability of fire spreading and spotting between subdomains is small, the probability of a rollback process 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 process synchronization. When the process synchronization 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 performance of the parallel ELFM code is significantly enhanced using the checkpointing and rollback techniques. Figure 9 shows the landscape pattern after a 1-year simulation period. The grey area in the landscape indicates that fires have occurred in that area.

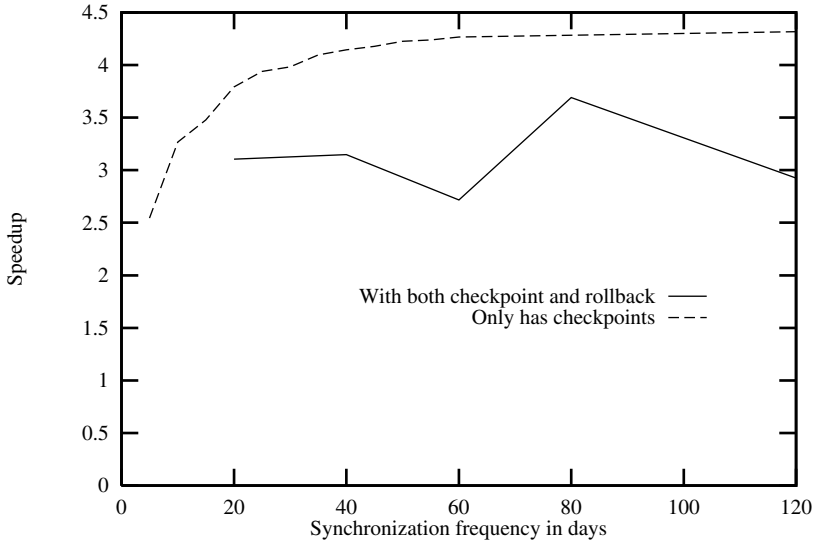

Fig. 8. Speedup of the parallel ELFM vs. synchronization frequency in a 4-workstation NOWs.

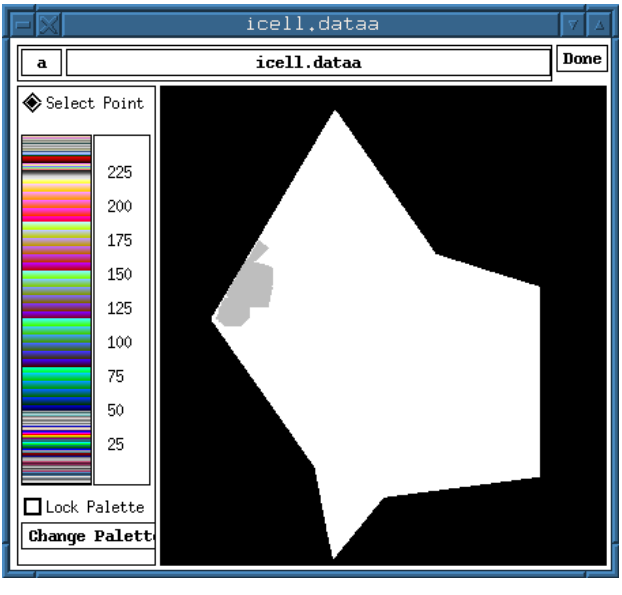

Fig. 9. Landscape pattern of WC2A in the Everglades after a 1-year period simulation by the parallel ELFM code.

6 Conclusion

In this paper, 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 simulation results show that a better speedup has been obtained compared to the parallel ELFM code without using the checkpointing and rollback techniques. The present study indicates that for certain type of parallel applications such as the ELFM, if the probability of interprocessor communication is small, checkpointing and rollback techniques can enhance their performance.

Our future work will focus on generalization of the parallel computation model with the mixture of a variety of asynchronous and synchronous computations. Parameters that affect the performance of the parallel applications, such as synchronization cost, asynchronous and synchronous computation ratio, load balancing, etc., will be studied both theoretically through numerical analysis and empirically through simulation.

References

1. D. K. Panda, and L. M. Ni. Special Issue on Workstation Clusters and Network-Based Computing. *Journal of Parallel and Distributed Computing*, **40**:1 – 3, 1997. 1, 2
2. F. He, 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, 3, 9, 11
3. L. T. Wille, and P. J. Ulintz. Parallel Simulations of Fire in the Everglades – Performance Analysis of Alogirthm. Report to South Florida Water Management District, Florida Atlantic University, December 1996. 2, 2, 11, 11
4. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, version 1.1. Available via anonymous ftp from *ftp.mcs.anl.gov*, June 1995. 2
5. P. Jalote. *Fault Tolerance in Distributed Systems.* Prentice Hall, Inc., Englewood Cliffs, NJ, 1994. 2
6. J. Wu. *Distributed System Design.* CRC Press, Boca Raton, FL, 1998. 4
7. Y.-M. Wang, 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, 1995. 4
8. Y. Wu, F. H. Sklar, K. Gopu, and K. Rutchey. Fire Simulations in the Everglades Landscape Using Parallel Programming. *Ecological Modelling*, **93**:113–124, 1996. 2