Using Time to Improve the Performance of Coordinated Checkpointing

Nuno Neves

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801

W. Kent Fuchs

School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-1285

Abstract

This paper describes and evaluates a coordinated checkpoint protocol that uses time to eliminate several performance overheads that are present in traditional protocols. The time-based protocol does not have to exchange coordination messages, does not need to add information to the processes' messages, and only accesses stable storage when checkpoints are saved. This protocol uses a simple initialization procedure to set checkpoint timers at the different processes. After the initialization, each process saves its state independently from the other processes. By disallowing processes from sending messages during an interval before the checkpoint time, the protocol prevents in-transit messages from occurring. Two coordinated checkpoint protocols were implemented on a CM5, and their performance was compared using several applications. Results showed that the time-based protocol outperforms the two-phase protocol in all applications.

1 Introduction

One effective way to recover distributed systems from failures is to use checkpointing and rollback recovery. Typically, a checkpoint protocol periodically saves the state of the application in stable storage. After a failure, the application rolls back to the last state that was saved and starts its re-execution. Checkpoint protocols are usually divided into two groups, uncoordinated [1, 9, 12, 18, 22] and coordinated [2, 5, 10, 11]. In uncoordinated checkpoint protocols, each process determines independently from the others the instant when its state should be saved. To avoid the domino effect, the checkpoint protocol also saves information about the messages that were exchanged among processes.

In coordinated checkpoint protocols, processes coordinate among themselves to determine which process states should be included in the application checkpoint. The coordination is necessary to guarantee that the application checkpoint is consistent and recoverable (section 3.2 explains these concepts). These protocols usually select one of the processes, the coordinator, to initiate the creation of the checkpoints and to ensure that each process saves its state [6, 14, 17, 19]. This task is accomplished with the exchange of a set of messages. The protocol adds information to each message to detect in-transit messages. Whenever an in-transit message arrives, the protocol saves it in stable storage, together with the state of the processes.

Both types of protocols have their own advantages. However, coordinated protocols have shown better performance than uncoordinated protocols when used with parallel applications [7]. Additionally, coordinated protocols do not need any piece-wise determinism assumption about the execution of the processes [8] and can tolerate failures that affect multiple processes simultaneously. Nevertheless, previous coordinated protocols have several overheads that should be avoided. In a typical coordinated protocol, the coordinator has to exchange three messages with each process. This overhead can become significant if the number of processes increases and the network is slow. The addition of information to messages to detect in-transit messages can also be an important overhead, depending on the level at which the protocol is implemented. If the protocol is implemented in a library, the overhead can be considerable, because each message might have to be copied. The third overhead is related to the in-transit messages. A process has to make an access to stable storage to save each in-transit message that it receives.

Previous time-based protocols [5, 15, 21] have used time to avoid the first overhead, the exchange of coordination messages. These protocols assume that processors' clocks are approximately synchronized, and use time to indirectly

Nuno Neves was supported in part by the Grant BD-3314-94, sponsored by the program PRAXIS XXI, Portugal. This research was supported in part by the Office of Naval Research under contract N00014-91-J-1283, and by the National Aeronautics and Space Administration (NASA) under Grant NASA NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS). This work used the CM5 computer system at the National Center for Supercomputing Applications, under the Grant CCR940004N.

coordinate the checkpoint creation. Each process saves its state whenever the local clock reaches checkpoint time. This paper presents a time-based protocol that also uses time to coordinate the checkpoint creation. However, this protocol does not rely on synchronized clocks. It uses a simple initiation procedure to set checkpoint timers at the different processes, and it saves new checkpoints whenever the timers expire. Contrary to the other time-based protocols, it also avoids the two other overheads that were mentioned previously. This is accomplished by preventing processes from sending messages that might become in-transit. The protocol defines a time interval before the checkpoint creation time during which processes are not allowed to send messages. The extent of the interval is proportional to the maximum message delivery time.

The proposed time-based protocol and a two-phase protocol were implemented on a CM5, and their performance was compared using several applications. Our results show that the time-based protocol outperforms the two-phase protocol in all applications. They also indicate that the most important overheads are the storing of the in-transit messages and the addition of information to messages. The less important overhead is the exchange of messages during the checkpoint creation.

2 Related Work

There is a broad literature in the subject of checkpointing and rollback recovery in distributed systems. Three different time-based checkpoint protocols have been previously proposed [5, 15, 21]. All these time-based protocols rely on approximately synchronized clocks to avoid exchanging coordination messages during the checkpoint creation. The protocol proposed by Tong et al. [21] creates checkpoints periodically, whenever the local clock arrives at the checkpoint time. The protocol assumes that a positive acknowledgment retransmission scheme is used to detect lost messages. A sender process keeps a copy of each message that it sends until it receives a positive acknowledgment. The checkpoint of a process includes all messages that have not been acknowledged. In-transit messages are detected by adding a checkpoint number to every message and its acknowledgment. A process saves in-transit messages as it receives them. Cristian and Jahanian [5] also use time to avoid message coordination. This protocol requires stricter assumptions about the synchronization of the clocks and assumes that messages' delivery times are bounded with high probability. However, it only needs to do one access to stable storage to save the in-transit messages. The protocol adds to each message the current checkpoint number and the time of the local clock. The checkpoint number is used to identify in-transit messages, and the local time is used to detect violations to the expected bounded delivery time. Ramanathan and Shin [15] proposed a time-based protocol in the context of the recovery blocks. This protocol relies on lock-step synchronized clocks and on the correct estimation of the expected time for a process to reach its acceptance tests. In this protocol, time was used to reduce the probability of blocking and to reduce the number of checkpoints that have to be kept.

The protocol presented in this paper does not necessitate synchronized clocks. It uses a simple initialization procedure to set the checkpoint timers. Contrary to the previous time-based protocols, it also prevents the existence of in-transit messages and does not need to add information to messages. Section 6 shows that these two overheads are more important than the actual exchange of coordination messages. The paper also presents the first implementation and experimental evaluation of time-based checkpointing.

Performance results of coordinated checkpointing have been presented by some authors in the past [3, 6, 7, 14]. These papers mainly show experimental results for the total execution of the checkpoint protocol. Our paper divides the checkpoint overhead into several categories, and presents experimental results for these specific categories. The following overheads are considered: message coordination, addition of information to messages, and in-transit message storing.

3 Distributed System Model

3.1 System Environment

The system is composed of a set of nodes interconnected by a network. A node contains a processing unit, a local memory and a local hardware clock. Clocks do *not* need to be synchronized among nodes. However, it is assumed that local clocks drift from real time with a maximum drift rate, ρ. This assumption implies that local clocks have at most an error of $\rho(e-s)$ seconds at the end of the real-time interval [s,e] seconds. The bounded drift rate condition applied to the local clock of a node n is

$$(1-\rho)(e-s) \leq C_n(e) - C_n(s) \leq (1+\rho)(e-s)$$

where $C_n(t)$ is time returned by the local clock when the real-time is t. The bounded drift equation can be used to derive a maximum deviation between the expiration of two timers. If two timers are started in two nodes exactly at the same time with the same initial value T, then they will expire by at most $2\rho T/(1-\rho^2)$ seconds from each other (we will approximate this value by $2\rho T$). Drift rates are in the order of 10^{-5} or 10^{-6} for most quartz clocks that are available in commodity computers, and for high precision clocks ρ is in the order of 10^{-7} or 10^{-8} [4].

Every node can store data in stable storage, and this data can be obtained after a failure by the correct nodes. Nodes fail according to the fail-stop model. In this model, a node affected by a fault stops its execution and remains stopped until recovery is initiated. All correct nodes can determine which nodes have failed.

Each node provides a computational environment for one or more processes. Each process executes a program and exchanges messages with the other processes. Messages are delivered in any order (no FIFO requirement) and communication channels can be unreliable, i.e., channels can lose or duplicate messages. However, if channels are unreliable, a simple mechanism based on sequence numbers and timeouts can be used to guarantee that a process receives all messages sent to it without duplicates. Messages are delivered to processes with a *bounded delivery time*, t_{dmax}. The total time to send a message, transmit the message through the network, and then receive the message is smaller than t_{dmax}.

3.2 Recoverable Consistent Checkpoints

A distributed application is executed by a set of processes that run on several nodes. The main responsibility of a coordinated checkpoint protocol is to save global states of the application. A global state includes the state of each process belonging to the application and possibly some messages. A process state *contains* the send event $send(m_i)$ if it has sent message m_i. A process state contains the receive event $rcv(m_i)$ if it has received message m_i. A generic coordinated protocol should record *recoverable consistent global states*, which satisfy the following two properties:

Consistency : If the global state includes a process state containing the receive event $rcv(m_i)$ then another process state must contain the corresponding send event $send(m_i)$.

Recoverability : If the global state includes a process state containing the send event $send(m_i)$ but no other process state contains the corresponding receive event $rcv(m_i)$ then the checkpoint protocol must save message m_i.

Global states saved by the checkpoint protocol are used to recover the application from failures that have affected one or more of its processes. Typically, the application rolls back to the last stored state and then starts to re-execute. The external results of the application re-execution should be equivalent to one of the results of a failure-free execution. This can only be accomplished if the application restarts from a global state that could have occurred during one execution with no failures [2]. For this reason, the global state can only contain receive events whose corresponding send events are also included. This characteristic is guaranteed by the consistency property. On the other hand, global states must include all messages that were in-transit at checkpoint time. Otherwise, these messages become lost during recovery because they are not re-sent by the processes. The recoverability property guarantees that all in-transit messages are available after the failure.

4 Time-Based Checkpoint Protocol

The time-based checkpoint protocol uses an *initialization procedure* to synchronize checkpoint timers and a *checkpoint creation procedure* to record recoverable consistent states of the application. The checkpoint creation procedure is executed periodically by each process whenever the local checkpoint timer expires. All processing is done locally without any exchanges of coordination messages. To guarantee that the consistency property is verified, the protocol disallows message sends during an interval after the expiration of the checkpoint timer. This interval is not constant, and increases as clocks drift apart. In an actual implementation, the blocking of message sends should not bring performance losses, because each process uses the interval to save its state. Timers are resynchronized when the interval becomes higher than the time taken to store a checkpoint. The recoverability property is ensured by preventing in-transit messages from occurring. The protocol disallows message sends during an interval before the checkpoint time. This interval is proportional to the maximum message delivery time.

The time-based protocol is described in the following way. First, we present the initialization procedure. Then, we derive two conditions, one that guarantees the consistency property and another that ensures the recoverability property.

4.1 Initialization Procedure

The initialization procedure initiates the processes' checkpoint timers in such a way that timers will expire within an interval of D seconds (if $\rho = 0$). Ideally, D should be made as small as possible, because that reduces the periods in which processes are not allowed to send messages (see next section). The initialization procedure is executed in three situations: to initialize the checkpoint protocol when the application starts, to initialize new processes that are added during the application execution, and to resynchronize the checkpoint timers. The resynchronization frequency depends on the value of the drift rate, but is relatively small.

The initialization procedure selects one of the processes to be the coordinator (the coordinator is usually the process that starts the application). The responsibility of the

```
Initialization:
  Coordinator:
    ckpTime = getTime() + T;
    setTimer(createNewCkp, ckpTime);
    setTimer(stopSMesg, ckpTime − (D + 2Tρ + t_dmax));
    while (TRUE) {
      time = getTime();
      broadcast(ckpTime − time);
      for each(p_i ∈ Processes) do receive(p_i);
      if ((getTime() − time) < (2 ∗ t_dmin + D)) {
        broadcast(FALSE);
        break;
      } else broadcast(TRUE);
    }
  Process i:
    continue = TRUE;
    while (continue) {
      receive(coord, interval);
      time = getTime();
      send(coord);
      receive(coord, continue);
    }
    ckpTime = time + interval − t_dmin;
    setTimer(createNewCkp, ckpTime);
    setTimer(stopSMesg, ckpTime − (D + 2Tρ + t_dmax));
```

Figure 1: Initialization procedure.

coordinator is to initiate the timers of the other processes. The coordinator cannot send an absolute time to the other processes, because clocks are not synchronized. It has to send a time interval. To calculate the interval $inter_c$, the coordinator subtracts its local time from the time when its timer expires. The timer at process p_i is set to $timer_i = currentTime_i + inter_c - t_{dmin}$ (where t_{dmin} is the minimum time to deliver a message).

There are several ways to distribute the time interval among the processes, and their complexity depends on the system that is being considered. Figure 1 shows one implementation of the initialization procedure. First, the coordinator adds to its local time the checkpoint period T[1] to obtain the first checkpoint time, $ckpTime$. Then, it sets two timers which will call the functions $createNewCkp$ and $stopSMesg$ (the next section explains the time values that were used), and broadcasts the interval. The other processes execute the code `Process i`. This code receives the interval and initiates two similar timers. Since different messages can experience distinct network delays, the coordinator loops sending the interval until it receives all answers within a time period smaller than $D + 2 * t_{dmin}$.

[1] For simplicity, checkpoints are created periodically with a constant period T. In a more general case, T can be different for each checkpoint as long as processes agree on the same value.

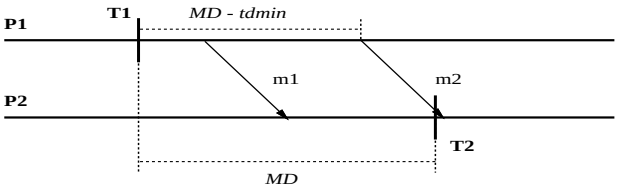

Figure 2: Consistency problem.

This guarantees that checkpoint timers will expire at most D seconds apart (if $\rho = 0$). In the experiments, D was set to 10 ms, which in most cases allowed the initialization of the timers in a single iteration.

4.2 Checkpoint Creation Procedure

4.2.1 Consistency

The checkpoint creation procedure has to save application states that verify the consistency property. Figure 2 shows an example of an execution that violates the consistency property. Process Pi saves its state whenever its checkpoint timer expires at Ti. Since timers are not exactly synchronized, process $P1$ sends a message $m1$ after saving its state, and $P2$ receives $m1$ before storing its state. The consistency property is violated because the global state contains $recv(m1)$ but does not include $send(m1)$. To avoid this problem, the time-based protocol disallows message sends during an interval after the checkpoint timer has expired.

Consistency condition : The nth application checkpoint satisfies the consistency property if no process is allowed to send messages $MD - t_{dmin}$ seconds after saving its nth checkpoint.

The consistency condition (CC) defines an interval in which processes can not send messages. This interval is equal to the *maximum deviation*, MD, between timers minus the minimum time required to deliver a message. Timers in different processes do not expire at the same time, because they are not exactly synchronized. The maximum deviation is the maximum time interval that can separate the expiration of any two timers, and is equal to $MD = D + 2nT\rho$. It depends on two quantities, the initial deviation between timers, D, and the clock drift since the last initialization, $2nT\rho$ (see bounded drift condition in section 3). The first quantity is constant, but the second one increases with time. This means that the amount of blocking can grow with time. However, MD increases slowly because drift rates are small. For instance, the initialization procedure can be used to start timers with $D = 10$ ms. If we assume a clock drift of $\rho = 10^{-6}$, then MD is equal to 100 ms after 12.5 hours.

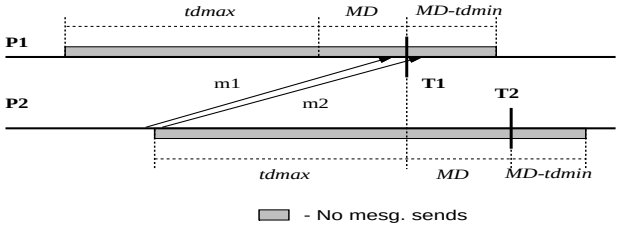

Figure 3: Total blocking interval.

The performance losses introduced by the CC condition are usually small in real systems. The CC condition does not prevent processes from continuing their execution. The CC condition only blocks a process if it attempts to send messages (actually, the process only has to block if it sends a synchronous message, because asynchronous messages can be queued). Also, the blocking interval can be used to save the processes' state. In current systems, disk accesses consume a large amount of time, which means that most or all MD time is used to store the process checkpoint (a typical process checkpoint takes at least 500 ms).

4.2.2 Recoverability

The easiest way to guarantee the recoverability property consists of avoiding the creation of messages that might become in-transit. This approach simplifies the implementation of the protocol during both the failure-free periods and recovery. The checkpoint protocol does not need to log any messages or to re-send or re-read messages. However, in this solution, processes can not send messages during an interval before their timers expire.

Recoverability condition : The nth application checkpoint satisfies the recoverability property if no process is allowed to send messages $MD + t_{dmax}$ seconds *before* its timer expires.

The example from Figure 3 can be used to illustrate the recoverability condition (RC). A message can become in-transit if it is sent before a process creates its checkpoint and is received after the other process has saved its state. In the figure, process $P2$ sends two messages, $m1$ and $m2$. Message $m1$ does not have to be stored, but message $m2$ would have to be saved if process $P2$ was allowed to send it. The maximum interval that prevents the existence of messages like $m2$ is equal to $MD + t_{dmax}$. The reader should note that RC does not prohibit processes from continuing their executions until they start to save their checkpoints. The process needs to block only if it attempts to send a synchronous message. If the message can be sent asynchronously, the process simply queues the message and continues with the computation. The message is sent after CC is verified.

```
stopSMesg:
  stopSendMesg = TRUE;
  setTimer(stopSMesg, ckpTime + T −
           (D + 2(CN + 1)Tρ + t_dmax));
createNewCkp:
  saveProcessState();
  CN = CN + 1;
  ckptTime = ckpTime + T;
  setTimer(createNewCkp, ckpTime);
  if ((getTime() − (ckpTime − T)) <=
           (D + 2(CN − 1)Tρ − t_dmin))
    resynchronizeTimers();
  stopSendMesg = FALSE;
  sendQueuedMessages();
```

Figure 4: Checkpoint creation procedure.

4.2.3 Creation of a Checkpoint

The time-based checkpoint protocol uses the CC and the RC conditions to create recoverable consistent checkpoints. The protocol can be implemented using the initialization procedure from Figure 1 and the checkpoint creation procedure from Figure 4. The creation procedure uses two timers: one that expires $MD + t_{dmax}$ seconds before checkpoint time, and another that expires at checkpoint time. Whenever the first timer terminates, it calls the function `stopSMesg`. This function sets a flag indicating that messages should be queued, and resets the timer. The function `createNewCkp` saves the process state, increments the checkpoint time by the checkpoint period T, and re-sets the timer. The variable CN counts the number of checkpoints that have been created since the last resynchronization. Then, `createNewCkp` tests the CC condition. If the condition is not verified, the function `resynchronizeTimers` is called to resynchronize the timers. This function sends a request for synchronization to the coordinator. The resynchronization procedure is similar to the initialization. Before returning, `createNewCkp` resets the flag and sends the messages that were queued.

5 Implementation

The experiments were performed on the CM5 of the National Center for Supercomputing Applications (NCSA). In this machine, each node contains a Sparc Cypress processor, 32 Mbytes of memory, and a network interface [20]. Nodes are connected by a control network and a data network. The control network supports communication patterns that involve all processing nodes (e.g., synchronization). The data network is used for point-to-point communications among nodes. The data network guarantees to each node an I/O bandwidth of at least 5Mbytes/sec (usual

bandwidths are between 8 and 10Mbytes/sec).

The time-based protocol and two versions of a two-phase protocol were implemented in a library that was linked with the applications. The library logically stays between the application and the CM5 communication library, CMMD. Whenever an application attempts to send a message, it first passes the message to the library. The library then executes all the steps required by the checkpoint protocol, and calls CMMD to send the message. The inverse procedure occurs on the receive side. This level of indirection allows a transparent implementation of the checkpoint protocols.

5.1 Two-Phase Protocol

The two-phase protocol utilized in the experiments is similar to the protocol that was used in a previous study on coordinated checkpoint protocols [6]. With the exception of the bounded delivery times, the time-based and the two-phase protocols make similar assumptions about the communication channels.

The two-phase protocol keeps in each process a checkpoint number counter, CN, that is incremented whenever a process saves its state. A new application checkpoint is processed in the following way. The coordinator increments its CN counter and broadcasts a special start message $< START, CN >$. After receiving the start message, a process updates the local CN and saves its state. Then, it sends a message $< SAVED, CN >$ to the coordinator. The coordinator, after saving its state and receiving all the $< SAVED, CN >$ messages, broadcasts the message $< END, CN >$ to terminate the checkpoint creation. To detect in-transit messages, the protocol tags each application message with the value of the local CN. A process stores in stable storage all messages that are received with a CN_m smaller than the current CN. A process initiates the creation of a new checkpoint if a message has a CN_m larger than the current CN. This last scenario can occur because channels are non-FIFO.

Two versions of two-phase checkpoint protocol were implemented. The first version follows exactly the procedure just described. In that version, the checkpoint library has to make an extra copy of each message passed by the application in order to add or remove the CN tag. The CMMD library does not provide a *gather send* function for noncontiguous buffers with different sizes. If this function were available, the copy could be avoided by saving in one buffer the CN tag and in another the message supplied by the application. In general, the extra copy has to be made in any system in which *gather send* functions are not available.

The second implementation avoids the copy in most cases, at the cost of making the protocol less general and less transparent. This implementation uses two bits of the application message tags to piggyback the CN. In this case, CN can only take four different values, 0 through 3. This optimization could only be used in a communication function that has a tag. In the other cases, it was necessary to make a copy. This restriction to the CN values requires the assumption that message delivery times are bounded and less than three times the checkpoint period. Also, the programmer can only use the first thirty bits of the message tag.

5.2 Time-Based Implementation

The time-based protocol makes the assumption that delivery times are bounded. This assumption can not be guaranteed in a system like the CM5, although it can be approximated. Our implementation uses two characteristics of the CM5 to approximate bounded delivery times. The first characteristic is the minimum I/O bandwidth guaranteed to each node. The second characteristic is the small number of processes that execute concurrently in each node. The following formula was used to calculate the maximum delivery time:

$$t_{dmax} = \frac{maxMesgSize}{5M} + extraTime$$

The first term in the formula corresponds to the maximum time that a message takes to be transmitted through the network. It depends on the size of the largest message that is sent by the application, $maxMesgSize$, and on the minimum bandwidth that is guaranteed by the network between two nodes. The second term is an upper bound on all the other delays that a message can experience. On some machines, it is difficult to determine this bound, because of the scheduling delays (e.g., a message can be received by the operating system, but the process is only scheduled after a long period of time). However, in the CM5 this problem is less important, because only a small number of processes are executed concurrently in each processor (usually only one or two processes). In the experiments, the value of t_{dmax} was set to 65 ms (25 ms to send the largest message and 40 ms for the extra time).

5.3 Applications

We used in the experiments six compute-intensive applications. These applications are either kernels of larger programs, or complete programs. Each application has different characteristics in terms of frequency of communication, amount of information exchanged, or pattern of communication. Table 1 presents the inputs that were used for each application. The applications were the following:

- `lu`: Performs the LU factorization of a matrix using the Gaussian elimination with partial pivoting. In each step, a node computes a column's multipliers and

Table 1: Description of the applications used in the experiments.

	Problem Description	Messages		
		Mesg./sec	KBytes/sec	Max. size
lu	50 512x512 matrices.	1763.1	3871.5	36864
mult	40 512x512 matrices.	6.0	200.8	34816
sor	3000 iterations 1024x1024 points.	748.6	3080.5	135432
tsp	10 problems with 20 cities.	143.2	586.6	4096
ga	population 1600, $4x10^6$ funct. evaluations	19.3	79.6	4096
ising	1200 iterations 1024x1024 grid.	329.2	1348.6	4096

broadcasts them to the other nodes. Next, all nodes update the remaining columns.

- `mult`: Multiplies several matrices by the same initial matrix A ($A * B_i = C_i, i = 1, 2, ...$). Each node starts with the same copy of a matrix, and calculates a few contiguous rows of the result matrix.

- `sor`: Uses the red-black successive overrelaxation iterative method to solve the Laplace equation. The problem is parallelized by giving to each node a certain number of contiguous rows of the resulting linear system of equations. In each iteration, a node calculates its points, and then exchanges the two boundary rows with two other nodes.

- `tsp`: Solves the traveling salesperson problem. A master node keeps a queue with a number of tours that still have to be evaluated. The other nodes request tours from the master and try to find a minimum length tour.

- `ga`: Is a parallel implementation of the genetic algorithm system GENESIS 5.0 [13]. `ga` solves a non-linear optimization problem. This application divides the initial population among the nodes. A node executes the genetic algorithm on its individuals, and after a few generations, it exchanges a few individuals with another two nodes.

- `ising`: Is a parallel simulation model of physical systems, such as alloys and polymers [16]. `ising` simulates in two dimensions the spin changes of Spin-glass particles at different temperatures.

6 Experimental Results

This section describes experiments made on a 32-node partition of a CM5. The results were obtained using the average of 3 to 5 experiments. In all runs, the execution times were on the order of 10 minutes, and the checkpoint interval was set to 1 minute. We did not use larger execution times because we had a limited amount of CM5 time that had to be used for both the code development and experiments. This, however, does not change our conclusions, because they are mainly comparisons between the two types of protocols. With larger checkpoint intervals we expect better performance for the time-based protocol and the two-phase protocol with the copy optimization, because they have small overheads between the creation of the checkpoints. The other version of the two-phase protocol will perform better or worse depending on the application. Since this paper does not propose new ways to reduce the checkpoint storing overhead, we do not include this overhead in the experimental results. We assume that both checkpoint protocols take the same time to store the application state.

6.1 Time-Based vs. Two-Phase

This section compares the time-based protocol with the version of the two-phase protocol without the message copy optimization. Table 2 presents the various results for executions with and without the checkpoint protocols. It is possible to observe that the time-based protocol performs better than the two-phase protocol in all applications. The time-based protocol shows overheads smaller than 1%, and the two-phase protocol has overheads between 1.5% and 13.1%. It is also possible to observe in the table that in most applications the two protocols took different numbers of checkpoints (due to the periodicity of the checkpoint creation). However, even if we calculate the overhead per checkpoint[2], the time-based protocol outperforms the two-phase protocol.

The time-based protocol achieves good results because it avoids most overheads while the application is executing.

[2]The reader should notice that this measure has to be used with caution, since the overhead of the checkpoint protocol does not occur only during the checkpoint creation, but also while the application is executing. In fact, a reasonable part of the overhead of the two-phase protocol occurs between the creation of the checkpoints.

Table 2: Experimental results on a 32-node partition of the CM5.

	No Ckp.	Time-Based				Two-Phase			
	sec	NCkp.	sec	%	Per Ckp. %	NCkp.	sec	%	Per Ckp. %
lu	451.5	7	455.5	0.9	0.13	8	510.6	13.1	1.64
mult	418.1	7	420.3	0.5	0.07	7	440.7	5.4	0.77
sor	497.5	8	499.6	0.4	0.05	9	547.7	10.1	1.12
tsp	449.3	7	450.3	0.2	0.03	7	462.9	3.0	0.43
ga	418.0	6	419.1	0.3	0.04	7	424.4	1.5	0.22
ising	467.0	7	471.8	1.0	0.15	8	492.4	5.4	0.68

The major cost of the protocol occurs only when the application is about to take its checkpoint. At checkpoint time, there is one interval in which processes can not send any messages. However, there is no blocking if the checkpoint happens to be taken in a computing phase of the process execution. Our experiments show that in most cases only a small number of processes had to block, usually less than 10%. Also, the amount of blocking depends on the instant when the process attempts to communicate. If the message is sent at the end of the interval, the process experiences almost no blocking.

The size of the blocking interval is proportional to clock drift and the maximum message delivery time. Since clock drifts are small, their contribution to the interval is usually modest. However, the term corresponding to the clock drift grows with time. As was mentioned previously, timers can be re-synchronized to solve this problem. It is more difficult to guarantee that the maximum delivery time will always be small (or that it even exists). For instance, it depends on the size of the largest message sent by the application and on the network bandwidth. The size of the messages can be kept within reasonable bounds if the checkpoint library implements message fragmentation. The applications that were used did not send messages larger than 135432 bytes, so there was no need to implement fragmentation.

6.2 Distribution of the Overheads

The performance costs introduced by the two-phase protocol can be divided mainly into message coordination, addition of information to messages, and in-transit message storage. Figure 5 presents, for each application, four bars showing how the costs have been sub-divided. The first bar shows the total overhead of the time-based protocol. The bar "No Copy, No In-transit" corresponds to the executions of the two-phase implementation with the no-message-copy optimization. This bar does not contain the time spent to save the in-transit messages (the function where the write occurs was commented), which means that

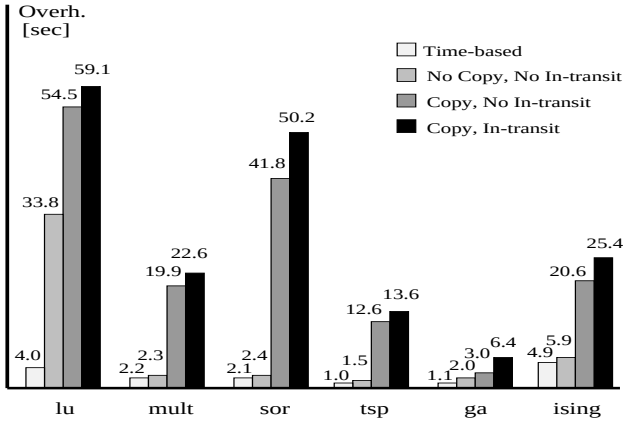

Figure 5: Time overheads of the checkpoint protocols.

it mainly shows the performance costs associated with the message coordination. It can be observed that the time-based protocol outperforms the optimized two-phase protocol in all applications. However, the overhead of message coordination is relatively small. The CM5 has a fast network, and we have implemented a completely asynchronous message coordination. No process is required to block while waiting for the coordination messages from the other processes. This type of implementation removes most of the message coordination costs. `lu` is the only application that shows a high overhead. In this application, it was not possible to remove all message copies. `lu` distributes the matrix multipliers through the nodes using the broadcast primitive of the CMMD. The broadcast function is synchronous and does not associate a tag with the messages. In this case, it was necessary to make a message copy to add the CN number. The `lu` bar contains copying overheads in addition to the message coordination.

The bar "Copy, No In-transit" corresponds to executions of the two-phase implementation that makes a message copy. This bar does not include the time taken to save the in-transit messages. It mainly shows the elapsed time

Table 3: In-transit messages.

	Number	Kbytes
lu	27	393.0
mult	11	405.5
sor	40	324.9
tsp	9	73.7
ga	18	150.2
ising	45	369.6

taken to make the message copies and to coordinate the checkpoint creation. The difference between the third and the second bar should be roughly equal to the time wasted while copying the messages. This time depends heavily on the application. Applications that exchange many and large messages have a higher copying overhead (see Table 1). The time also depends on the communication pattern that is used by the application. `tsp` sends a larger amount of information per second than `mult`, but it has a smaller copy overhead. In the `mult` application, nodes send to the master the computed rows, and then wait for new rows of another matrix. The master only distributes new work after receiving all the previous rows. This means that each message copy made by the master not only makes the receiving process wait, but also all the other processes that are expecting new rows. Communication is less synchronous on `tsp`. A node sends its results to the master, and then receives new work without having to wait for the other nodes.

The last bar adds to the third bar the time required to save the in-transit messages. These values are the average of the 3 best execution times of 5 experiments. The number of in-transit messages that have to be stored can change dramatically from one run to the next. For instance, with the `ga` application, there was an execution for which it was necessary to save 207 messages with a total size of 1.7 Mbytes. These values are quite different from the averages shown in Table 3. The in-transit storage overhead depends on several factors, such as the number of in-transit messages, the size of the messages, and disk contention while the writes are being done. It also depends on the type of communication that is used by the applications. As was mentioned previously, if several nodes are expecting to receive a message from the same node at the same time, a disk access made by the sender can make several nodes wait. The use of different communication primitives can change the probability that in-transit messages will occur. For instance, `lu` sends more messages than `sor`, but has a smaller number of in-transit messages (see Tables 1 and 3). This is because `lu` uses a synchronous broadcast primitive.

7 Conclusions

The paper presented a checkpoint protocol that uses time to avoid most performance penalties introduced by traditional coordinated protocols. The protocol does not rely on approximately synchronized clocks to eliminate the message coordination overhead. It uses a simple initialization procedure to start the checkpoint timers. Contrary to previous time-based protocols, it also eliminates the overheads of in-transit message storage and addition of information to messages. This is accomplished by preventing processes from sending messages during an interval before the checkpoint time.

Experimental results were presented showing that the time-based protocol outperforms a two-phase protocol. Results also show that message coordination overhead is less important than the overhead of writing in-transit messages to stable storage and the overhead of adding information to messages.

Acknowledgments

We would like to thank the anonymous referees for their suggestions. We also wish to thank Jenny Applequist for her comments that helped to improve the readability of the paper.

References

[1] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[2] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[3] T. Chiueh and P. Deng. Efficient checkpoint mechanisms for massively parallel machines. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing*, June 1996.

[4] F. Cristian and C. Fetzer. Probabilistic internal clock synchronization. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pages 22–31, October 1994.

[5] F. Cristian and F. Jahanian. A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, pages 12–20, September 1991.

[6] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[7] E. N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, pages 298–307, August 1994.

[8] A. Goldberg, A. Gopal, K. Li, R. Strom, and D. Bacon. Transparent recovery of Mach applications. In *Proceedings of the Usenix Mach Workshop*, pages 169–184, July 1990.

[9] D. B. Johnson and W. Zwaenepoel. Sender-based message logging. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, pages 14–19, July 1987.

[10] J. L. Kim and T. Park. An efficient protocol for checkpointing recovery in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):231–240, August 1993.

[11] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[12] N. Neves, M. Castro, and P. Guedes. A checkpoint protocol for an entry consistent shared memory system. In *Proceedings of the Thirteenth Annual Symposium on Principles of Distributed Systems*, pages 121–129, August 1994.

[13] N. Neves, A.-T. Nguyen, and E. L. Torres. A study of a non-linear optimization problem using a distributed genetic algorithm. In *Proceedings of the International Conference on Parallel Processing*, August 1996.

[14] J. S. Plank. *Efficient checkpointing on MIMD architectures*. Ph.D. thesis, Princeton University, June 1993.

[15] P. Ramanathan and K. G. Shin. Use of common time base for checkpointing and rollback recovery in a distributed system. *IEEE Transactions on Software Engineering*, 19(6):571–583, June 1993.

[16] J. G. Silva, J. Carreira, H. Madeira, D. Costa, and F. Moreira. Experimental assessment of parallel systems. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing*, June 1996.

[17] L. M. Silva and J. G. Silva. Global checkpointing for distributed programs. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 155–162, October 1992.

[18] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[19] Y. Tamir and C. H. Séquin. Error recovery in multicomputers using global checkpoints. In *Proceedings of the International Conference on Parallel Processing*, pages 32–41, August 1984.

[20] Thinking Machines Corporation. *Connection machine CM-5 technical summary*, November 1993.

[21] Z. Tong, R. Y. Kain, and W. T. Tsai. A low overhead checkpointing and rollback recovery scheme for distributed systems. In *Proceedings of the 8th Symposium on Reliable Distributed Systems*, pages 12–20, October 1989.

[22] Y.-M. Wang and W. K. Fuchs. Lazy checkpoint coordination for bounding rollback propagation. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 86–95, October 1993.