Implementing and Evaluating Automatic Checkpointing

Antonio S. Martins Jr.
State University of Maringá
Data Processing Center
Av. Colombo, 5790
87020-900 – Brazil
+55 (44) 32614014
asmartins@uem.br

Ronaldo A. L. Gonçalves
State University of Maringá
Informatics Department
Av. Colombo, 5790
87020-900 – Brazil
+55(44) 32614071
ronaldo@din.uem.br

Abstract

As the size and popularity of computer clusters go on growing, fault tolerance is becoming a crucial factor to ensure high performance and reliability for applications. To provide this facility, a checkpoint mechanism is used to recover a failed parallel application rolling it back to an execution moment prior to occurrence of the failure. In this work we present a mechanism for managing checkpoint operations during the failures automatically. This mechanism records periodically the application's context, identifies failed nodes and restarts MPI processes on the remaining nodes, allowing the continuity of the application and taking advantage of the computing accomplished previously. We describe a lot of changes inside source of the LAM/MPI. Experiments with an application for recognizing DNA similarity showed that despite the overhead caused by periodic checkpoints, the benefits can reach about 50% on a small cluster.

1. Introduction

As the human knowledge increases more complex problems appear. For many computational problems, the complexity is usually measured by the amount of calculations necessary to solve them. Thus, as bigger the problem, more time is need to solve it. As examples, we have the calculations for climatologic forecast, analysis of satellite images, aerodynamic simulations, pattern recognition, seismic analysis and the genome sequencing.

Parallel processing can be used efficiently to solve these problems. It is done dividing the complex applications in a group of smaller and simpler tasks, so that they can be executed simultaneously by independent processors. However, the construction of specific parallel machines has high cost, a million dollars for gigaflop approximately.

Clusters of the Beowulf class [1,2] have been used as alternative machine with a great advantage: relatively cheap conventional computers are connected in order to work in parallel on the solution of the same problem. In this model, as bigger is the number of nodes, bigger is the potential for parallel processing.

However, with the increase of the amount of nodes, the reliability decreases because the occurrence of failures is directly proportional, restricting the scalability of cluster [3]. To minimize this problem, a checkpoint mechanism is used to record the context of an application during its execution for later recover it in the case of failures. However, in many MPI implementations, this mechanism depends on the user's interventions to activate the checkpoint operations, such as LAM/MPI.

The present paper, started in [22], proposes the automation of the checkpoint operations on the LAM/MPI, implicitly for the users. Changes were made in some functions of the LAM/MPI library, including code corrections. Our automatic checkpointing was experimented with a parallel application for DNA similarity recognition on a cluster of 8 nodes. This paper is an improved version of a previous work [30], which was published in Portuguese and experimented with a mathematical application on a cluster of 4 nodes.

This paper is organized as follows. Section 2 presents an overview on the checkpointing. Section 3 briefs the features of the LAM/MPI. Section 4 presents the proposal showing how it was implemented. Section 5 presents the performance evaluation applied on a DNA-based application. Conclusions and references appear in the sections 6 and the last one, respectively.

2. Checkpoint Mechanism

The goal of the checkpointing is to establish a recovery point in the application execution, recording enough information of its context for later recover it (roll-back

1-4244-0910-1/07/$20.00 ©2007 IEEE.

recovery) in case of failures, minimizing the work loss [4,5] already accomplished. Checkpointing provide support for many other mechanisms like fault tolerance, process migration, load balance, task swap and code debug [4,6,7].

When a process is running its context is composed by memory and registers contents and by the context of the operating system (included file system). Usually, the memory is organized in four segments: code, variables, heap and stack. During the checkpointing the code is not usually recorded, because it can be recovered from the source file.

When a checkpoint mechanism is implemented in the kernel's level, the private information of the operating system, necessary to execute the application, can be easily obtained during the roll-back recovery. However, when this mechanism is implemented in user's level, that information needs to be recorded in the checkpoint container to be accessed later [4,6]. Besides the internal structures of the operating system and file system, information about the user interface (graphical or text), external servers and others related to the execution environment may need be recovered. Many checkpoint systems supply primitives for programmers, allowing the control of the information that they want to record. However, they have all responsibility by the reconstruction of the execution context during the roll-back recovery.

Checkpointing in distributed memory systems is more complex, because it is necessary to keep a coherent global state of all system to ensure a safe recovery. Figure 1 sketches situations about this question on a multiprocessor. The horizontal lines represent independent executions of the processes A, B and C. The stars represent checkpoints associated to each process and the arrows from m1 to m4 represent transmitted messages among remote processes.

Figure 1 shows there are sets of the checkpoints that do not ensure a safe recovery. For example, the set of checkpoints composed by A2, B2 and C2 is not coherent one due to the message m2, because if the execution is recovered in these points, the process B will send m2 again, which has already been received by the process C. On the other hand, the checkpoints composed by A3, B3 and C2 provide a consistent global state, with no lost or duplicated messages.

The problem is each process creates periodic checkpoints without coordination, making hard or unfeasible in some cases the recovery of a coherent global state. The called domino effect can appear when the processes create multiple checkpoints as follows. During a failure, the checkpointing will try to recover the application starting from the most recent checkpoint on each process. However, when it is not possible, the system will try again and again while the operation to fail. The attempts will continue successively until the last possibility, returning back to the initial point of the execution, because no consistent global state was reached [6,8]. However, there are two forms for handling the domino effect: coordinated checkpointing and checkpointing with message logs.

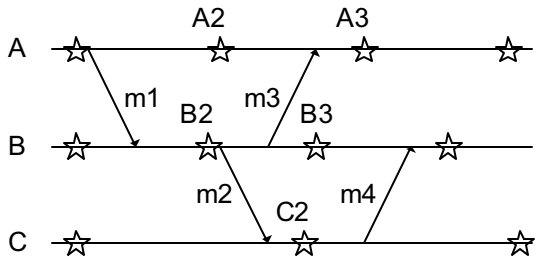

Figure 1. Checkpointing on a Multiprocessor

With coordinated checkpointing, the processes cooperate to execute all checkpoints synchronously, ensuring that all sent messages were received before the checkpoint. With this approach, all messages that still were not consumed will be sent and received also naturally. Obviously, it is possible that some processes execute the same lot of instructions again, but no resent message will be lost. A great advantage of this model is that any failure causes that all processes needs just the most recent checkpoints, reducing the storage size. Another advantage is the algorithm for recovery is trivial. Other models request complex algorithms to recover the application [6,9,10,11].

Using checkpointing with message logs, a specific control will exist exclusively for the messages. However, the message flow must be fully deterministic, that is, starting from a given checkpoint the program will always produce and consume the same messages. In this scenario, a process can record all the received messages after the last checkpoint and thus rebuild any safe state. Algorithms for message logs work so, allowing independent checkpoints [6,8].

3. About the LAM/MPI

The Message Passing Interface (MPI) is a de facto standard for parallel programming based in message passing for large scale distributed systems. Implementations of MPI support the middleware layer for many high-performance applications [3]. However, the MPI standard itself does not specify any kind of fault tolerance. In addition, the most widely used MPI implementations have not been designed to be fault tolerant. LAM/MPI is a well known MPI implementation. Some projects were developed with the intention of extending pre-existent MPI implementations including fault tolerance. COCHECK [12], CLIP [13], MPICH-V [15] and LAM/MPI-CR [3,16,17] are examples. We do not have knowledge that they have automatic checkpointing/restart.

3.1. Organization

LAM/MPI [3] uses a small user's space daemon for the control of processes, communication and I/O redirection. This daemon, called lamd, is launched during the booting of the MPI platform. All daemons lamd together form the base of the execution environment of LAM/MPI [16,17,18].

The user interface is constituted by several command line applications (MPI commands), which are responsible by necessary actions to creation and execution of distributed applications. The three main MPI commands are: 1) lamboot, which is the responsible by loading the daemon lamd and initialization of the execution environment; 2) mpicc, mpic++ and mpif77, which are compilers/linkers that allow to the users write applications in languages C, C++ and Fortran, respectively; 3) mpirun, which is the responsible for loading and controlling the execution of the application.

The execution environment of distributed applications on LAM/MPI can be exemplified in the Figure 2. We may see the MPI application is composed by three processes executing on an ideal cluster composed by three nodes. It is known that LAM/MPI creates a logical cluster on the physical nodes where the lamd daemons are loaded. The MPI application is executed on that logical cluster. Each physical node when executing a copy of the daemon lamd, becomes a logical node of the LAM/MPI cluster.

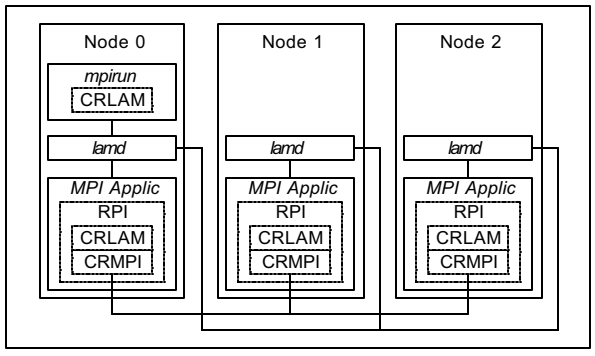

Figure 2. A Distributed Application on the LAM/MPI

The mpirun program is executed in the master node. During the loading of application processes, it is possible to select the 1) MPI module, that implements the communication primitives, like send and recv, for using of users' applications and the 2) RPI module (Request Progression Interface) [20], that implements the dependent of hardware communication basic layer (TCP, Myrinet, SMP) and some other functionalities, including checkpointing.

The module RPI, coded as crtcp, is composed by the modules CRLAM and CRMPI, which implement the checkpoint bookmark routines. The CRMPI implements the communication primitives and the CRLAM implements the coordination among the processes of the MPI application. The lamd daemon also provides a resource for fault tolerance through a heartbeat routine, which tests the connectivity of the cluster nodes and updates the environment in the case of failure in some node.

3.2. Checkpointing

The BLCR (Berkeley Labs Check-point/Restart) package [7] was developed to support interactive checkpointing on the LAM/MPI [3]. It provides the commands cr_checkpoint and cr_restart, which can be used by the user explicitly. The checkpointing steps during the context recordings are quite simple and they can be summarized as follows:

- **user**: calls the command cr_checkpoint passing the pid of the mpirun process.
- **mpirun**: receives the user´s request (via CRLAM).
- **mpirun**: propagates the request for each MPI process (via CRMPI).
- **each MPI process**: dialogues with the other processes in order to establish a consistent global state.
- **each MPI process**: calls also the command cr_checkpoint passing its own pid.
- **cr_checkpoint**: records the context of each MPI process.
- **each MPI process**: continues its normal execution after the cr_checkpoint return.
- **mpirun**: prepares itself for a possible recovery and replies to the command cr_checkpoint that it is ready.
- **cr_checkpoint**: records the context of the mpirun process.
- **mpirun**: continues its normal execution.

The sequence of steps of the recovery algorithm can also be summarized as follows:

- **user**: calls the command cr_restart passing the context file name associated to mpirun created by cr_checkpoint.
- **mpirun**: re-executes itself (via execve system call) passing the configuration file prepared by itself. This file informs that the mpirun must to recover the MPI application.
- **mpirun**: executes from the recovered context (after the checkpointing call in the code): reads the configuration file and requests the re-execution of the MPI processes using the cr_restart command followed by the name of context file associated to each one created by cr_checkpoint.
- **each MPI process**: executes from the recovered context (after the checkpointing call in the code).
- **each MPI process**: sends its "updated" local information (_gps structures) to mpirun.
- **mpirun**: builds the global information table and propagate it back to all MPI processes.
- **each MPI process**: receives the information about the others MPI processes and rebuilds the communication channels with everyone.
- **each MPI process**: continues its normal execution.
- **mpirun**: continues its normal execution.

4. Automatic Operations

This work presents the design and implementation of the mechanism to automate checkpoint operations on the LAM/MPI, including context recording, failure detection and application recovery, using BLCR [3,7,21].

4.1. Recording

Our mechanism generates recovery points automatically (checkpoints) in a frequency predefined by the user and also during the MPI collective function calls [19]. The implementation of the automatic checkpoint operations is based on threads created in runtime, which execute the new functionalities inserted into LAM/MPI. We have implemented two threads: cr_ckpt_time() and cr_ckpt_coll(). The simplified algorithms of these threads can be observed in the Figure 3 and Figure 4.

The thread cr_ckpt_time calls the command cr_checkpoint, in intervals defined by the user. The thread cr_ckpt_coll calls the command cr_checkpoint during the execution of collective MPI functions that involve synchronization, as the functions implemented in the module CRMPI: MPI_Barrier, MPI_Scatter and MPI_Gather, which also were adapted. The combined use of these threads ensures the creation of coherent checkpoints on the LAM/MPI.

```
cr_ckpt_time()
{ ckpt_time = now + interval;
  while (TRUE)
  { wait(ckpt_time - now);
    lock(mutex);
    if (ckpt_time <= now)
    // it was not done by cr_ckpt_coll
    { cr_checkpoint; //takes a delay
      ckpt_time = now + interval + delay;
    }
    // else does nothing because it was already
    // done by cr_ckpt_coll
    unlock(mutex);
  }
}
```

Figure 3. Thread cr_ckpt_time()

The main modification in the collective functions is the insertion of a call to primitive send, in order to allow that they can do a checkpoint recording request to thread cr_ckpt_coll located in the mpirun. This call is placed where the processes are synchronized waiting the finalization of the collective function. For example, the original MPI_Barrier works as follows. There is a root process, which is the responsible process for the synchronization. All the other processes send messages to it through the MPI_Barrier function and stay blocked waiting a reply. When all the messages are received, the root process replies to all unblocking them. In the new MPI_Barrier, before sending the reply for unblocking a checkpoint recording is done, exactly when all of the processes are blocked.

```
cr_ckpt_coll()
{ while (TRUE)
  if (recv(checkpoint_request))
  { lock(mutex);
    if ((ckpt_time - delay) <= now)
    // an interval elapsed with no checkpoint
    {  cr_checkpoint; //takes a delay
       ckpt_time = now + interval + delay;
    // else an interval still didnt elapse
    // after the last checkpoint
    }
    unlock(mutex);
  }
}
```

Figure 4. Thread cr_ckpt_coll()

The synchronization among the checkpoint threads is controlled by mutex mechanism, which ensures the mutual exclusion during concurrent checkpoints, as well as the updating of the shared time control variables. A heuristic updates the intervals among the checkpoints, which are calculated using the average time spent during checkpoints added to the interval defined by user. The activation moment of these threads may be seen in the area "B" of the Figure 5. This figure shows the simplified algorithm of the mpirun, where the delimited areas indicate the modifications.

4.2. Fault-Detection and Recovery

The fault detection and automatic recovery are implemented by a routine inserted in the mpirun that is able to recognize failures among the cluster nodes by asking for their life to each lamd daemon (fault detection by heartbeat), which must be executing in the safe mode. This routine, called cr_wait, substitutes the original routine, called rpwait.

The rpwait waits the finalization of the MPI processes in a blocking call to the function nrecv. Instead of this, the cr_wait uses a semi-busy wait (loop with sleep), where the function nprobe is used to verify the existence of messages from lamd daemons. If no message exists from some lamd daemon, a query to that lamd is done. If no reply is received, it is supposed a failure in the associated node. In this case, the mpirun restarts the MPI application, from the last checkpoint, on the remaining nodes. This routine may be seen in the area "C" of the Figure 5.

4.3. Process Migration

When the MPI processes connect to the LAM/MPI environment, during the execution of the MPI function MPI_Init, they create a communication channel using a pipe with each local lamd daemon. The name and location of this communication channel are dependent of the

physical node where each process is executing. During the checkpoint, the name of this pipe is stored in each MPI process context in order to be available during the recovery. However, this information allows the re-establishment of the communication just over the same physical nodes. Therefore, during the process migration among physical nodes, the name of the recovered pipe becomes incorrect and the communication is not possible. To solve this problem, which was not handled in [3], the pipe name is updated before to call the routine responsible for the communication reestablishment as follows.

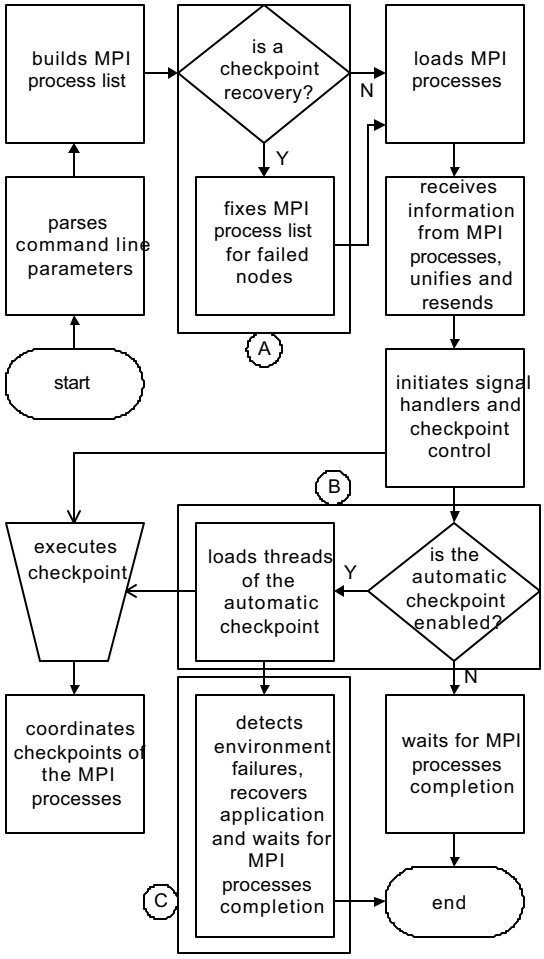

Figure 5. Changes in the mpirun application

Each MPI process keeps information about the other processes in a structure called _proc, which can be accessed using a substructure _gps as search key. The substructure _gps is composed by 4 identifiers: the logical node (gps_node), the process in the Unix environment (gps_pid), the process in the logical node (gps_idx) and the process in the MPI application (gps_grank). After the restart of the MPI processes, the information about the logical and physical nodes of each process is updated in the _gps structure, allowing the re-establishment of the communication channels even that any process has been migrated. The routine lb_fault_app was developed, inside the mpirun, to redistribute the MPI processes among the remaining nodes. This routine acts in the area "A" in the Figure 5.

5. Performance Evaluation

We have accomplished a lot of practical experiments in order to evaluate the performance of the automatic checkpointing. To induce failures we turn off one or more processors manually during the execution of the MPI application. The failures were induced in different moments, after the application to have already been executed a certain percentage of its lifetime. We have experimented on a cluster of PCs running our extended LAM/MPI, which we have called "Cluster-8". This cluster has 1 master and 8 slaves. Each one has an IDE disk and a Fast-Ethernet board. The master is an Atlhon 1.0GHz with 256 MB of RAM. The slaves are 4 Pentium IV 1.8GHz and 4 Pentium IV 3.0GHz, each one with 512MB of RAM. The slaves boot locally, but they use the "/home" of the master through NFS.

To analyze the effects under failures we executed a MPI application provided partially by Alves [26] and adjusted according to Needleman & Wunsch [23], Smith & Waterman [24] and Gotoh [25]. This application is able to determinate the similarity among DNA sequences parallely. We refer to it as DNA/MPI Application. The DNA/MPI Application was applied on the bacterias Xac (Xanthomonas axonopodis) and Xcc (Xanthomonas campestris), using the same idea used by Almeida [27]. Xac and Xcc were obtained in the GenBank and they have 5,175,554 and 5,076,187 nucleotides, respectively. However, we have worked in terms of amino acids, which are groups of 3 nucleotides. This approach reduces the number of components to be compared despite of increasing the number of comparisons because there are 20 different types of amino acids against just 4 different types of nucleotides.

The similarity is measured in a score, which is obtained by comparing the two sequences of DNA (strings for us) component to component. The basic idea to score the two strings is as follows. Consider a column i consisting of symbols X[i] and Y[i]. If X[i]=Y[i], this column will receive a partial score p(X[i],Y[i])>0 (1 in this example), otherwise, it will receive a partial score p(X[i],Y[i])<=0 (0 in this example). Finally, a column with a blank will receive a partial score p(X[i],Y[i])=–c, where c is in ℕ (0 in this example). The final score is the sum of these partial scores. Then, we suppose the better alignment is that one with maximum score, which is called similarity. The matrix S, where each element S[i,j] represents the similarity among Y[1..i] and X[1..j], can be used to facilitate the matching algorithm.

```
if (myrank == 0) //I am the master
{  GetSequences(X, sizex, Y, sizey);
   //I send data to slaves
   MPI_Broadcast(sizex, sizey);
   MPI_Broadcast(X, sizex);
   slicey=sizey/nproc;
   //I send a different slice of Y to each slave
   for (slave=1; slave <=nproc-1; slave++)
      MPI_Send(Y[slave*slicey], slicey, slave);
}
else //I am a slave
{  //I receive data from the master
   MPI_Broadcast(sizex, sizey);
   slicey=sizey/nproc;
   MPI_Broadcast(X, sizex);
   //I receive my slice of Y from the master
   MPI_Recv(Y, slicey, 0);
}
//I create a partial matrix S
S=CreateMatrix(Y, slicey, X, sizex);
slicex=sizex/nproc;
for (k=0; k<=sizex; k:=k+slicex)
{  if (myrank!=0)
      //I receive in S[0] data from the previous process
      MPI_Recv(S[0], slicex, myrank-1);
   for (j=k+1; j<=k+slicex; j++)
      for (i=1; i<=slicey; i++)
         S[i][j]=max(S[i-1][j]-c, S[i][j-1]-c,
                     S[i-1][j-1]+p(Y[i], X[j]));
   If (myrank!=nproc-1) //I am not the last
      MPI_Send(S[slicey], slicex, myrank+1);
}
```

Figure 6. Simplified DNA/MPI Application

Figure 6 shows the simplified algorithm of the DNA/MPI Application – some statements and parameters are omitted – which works as follows. There are nproc processes, including the master. Firstly, the master reads the sizes and the sequences X and Y. Then, it sends the sizes and X for all slaves, using the function MPI_Broadcast. Notice that this function can be used for both sending and receiving. After that, Y is split in nproc subsequent slices, which are distributed one for each process, using the functions MPI_Send/MPI_Recv. Each slave creates part of the matrix S and calculates the scores associated to it. After the calculation of each subpart, the process sends the last line to the following process (myrank+1). This technique, known as "wavefront", is described in [26].

5.1. Analysis of the Results

In our experiments we executed the DNA/MPI Application on the whole genome of the bacteria Xac, which has 1,432,518 amino acids. However, we used just a part of the genome of the bacteria Xcc containing 10,574 amino acids. The results are plotted in the Figure 7 and Figure 8. Failures were induced on 1 up to 4 slaves during the execution of the application. After the failure detection the application was re-executed on the remaining nodes. The failures were induced on the 4 fastest nodes firstly and after on the 4 slowest nodes. The average results were considered as final. These figures show two different situations: "Normal n" and "ACR n".

The label "Normal n" represents the variation of the execution total time of the DNA/MPI application with no checkpoint mechanism but under failures of n nodes after x% of its execution. We suppose that the operator of the application will perceive immediately the failures and will restart the application with no waste of time with the slaughter of the alive processes. This is an optimistic situation.

The label "ACR n" (acronym for Automatic Checkpoint/Restart) represents the same execution but with our automatic checkpoint mechanism. Unlike of the first one, this is a realistic situation because it counts the waste of time with the detection, killing of the alive processes and the recovery on the remaining nodes, everything automatically. Besides, in this situation, the processes need to establish a new dialog about theirs new positions before of restarting the execution.

In the Figure 7, the lines "Normal n" are the 4 above in the right side and the "ACR n" are the others below. The y axis show the total time (lifetime) of the execution of the application in seconds and the x axis show the percentages of executed code before the failure (from 10% to 90%), except at the start of the lines (labeled "no failure"), which show the time of execution with no failure. In this point we may see the ACR causes an overhead in the application. This overhead is due to the heartbeat mechanism to detect failures and to the periodic recordings (each 1,000 seconds) of the process contexts of the DNA/MPI application. In these experiments, the overhead was about 13,6% (5,098 against 4,887), but it is paid back quickly when failures happen.

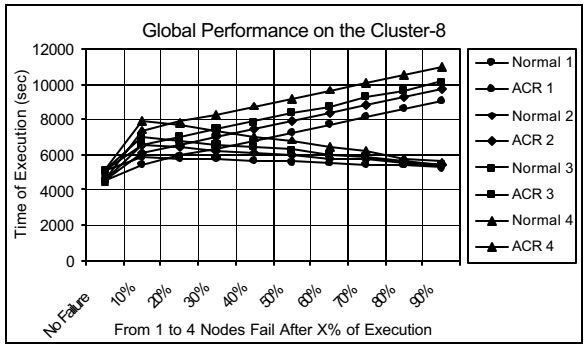

Figure 7. Performance ACR and Normal on 8 Nodes

Still in this figure, notice that as larger is the elapsed execution time before the failures, larger is the difference between the Normal and ACR. In fact, these two lines move forward in vertically opposed directions, increasing the Normal and decreasing the ACR. Also, even using ACR, the execution time of the application can to increase so much when one or more nodes fail. However, this fact happens because the application is restarted on fewer nodes than before. Thus, the remaining nodes will execute more than one process. Besides, in our application the processes could be classified as cpu-bound. This situation

can to multiply the remaining execution time because the global time of the application will be tied in the time of the slowest node, not improving anything if the fastest nodes finish first.

We may see that the increase in the number of failures causes the increase of the execution time. For the Normal situations, this increase arises as a lateral displacement almost constant in the whole course of the graph lines. In a superficial analysis, this fact seems strange because when already there is at least one node with two processes, the execution time should not increase if other nodes also execute 2 processes since that the time of the slowest node basically defines the time of the application.

However, the specificities of the application in conjunction with the operating system policies must be considered either as advantage or as damage in the time increase. In our application for example, the processes exchanges the partial results periodically. Thus, when 2 processes are placed in the same node, the remote message passing must be serialized because of the I/O concurrence and this fact can delay the application. For the execution with ACR, this increase is smaller when the failures happen after the application to be executed a larger time.

Therefore, the "ACR n" lines converge in the end of the execution. It is a comprehensible behavior because it is presumed that in the limit of 100% of execution, the time impact caused by the recovering of the application with ACR is not much more than the overhead already mentioned previously. In this situation, the dependencies among processes in terms of message passing already were solved. Figure 8 shows the speedups for different situations.

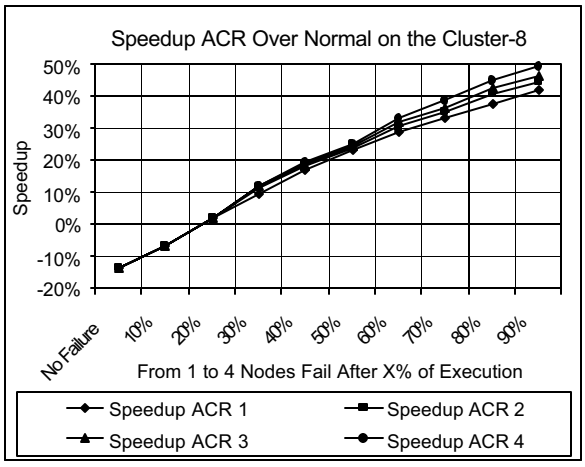

Figure 8. Speedup ACR over Normal on 8 Nodes

The highest speedup reaches around 50%. The DNA/MPI application was split in 8 parts, taking advantage of the time-slice given by the operating system if compared with another situation does not related here where the application is split in 4 parts. The application gets the processor more times. In fact, developing an application with more processes can be a technique to deceive the operating system in order to get more cpu.

6. Conclusions and Future Works

In this work we present an automatic checkpointing proposed for extending the LAM/MPI using a basic infrastructure provided by BLCR. Also, we describe a lot of changes inside the open source of the LAM/MPI and experiment on a parallel application for recognizing of DNA similarity. The results show that despite the little overhead caused by periodical checkpoints, the benefits with our mechanism can reach about 50% on a cluster of 8 nodes, when compared with the necessary time for the re-execution of the same application running without checkpointing. Detecting and recovering the application automatically after failures reduces significantly the waste of time with both the wait for the operator's intervention and with the processing already accomplished.

As future works, the use of the local disks to record the process contexts will be better investigated to reduce the traffic in the interconnection network and the overhead in the file server, increasing the checkpointing performance. Besides, we will be combining this mechanism with techniques for load balancing in MPI clusters, improving the performance after the recovery by allocating processes on more appropriated nodes. Anyway, the unequivocal conclusion is: the use of automatic checkpointing in LAM/MPI is a fundamental support to provide reliability to high performance computing.

7. Acknowledges

Our thanks to CNPq, CAPES and FUNDAÇÃO ARAUCÁRIA.

8. References

[1] BUYYA, R. "High Performance Cluster Computing: Architectures and Systems". V.1. N.J. Prentice-Hall, 1999.

[2] STERLING, T. "Beowulf Breakthroughs – The Genesis of Linux Clusters in High Performance Computing". Linux Magazine. Jun. 2003.

[3] SANKARAN, S. et al. "The LAM/MPI Check-point/ Restart Framework: System-Initiated Check-point". In: Proceedings of LACSI Symposium. Santa Fé, USA. 2003.

[4] WANG, Y-M.; et al. "Checkpointing and its Applications". In: 25th International Symposium on Fault-Tolerant Computing, Pasadena. 1995.

[5] PLANK, J. S.; et al. "Libckpt: Transparent Checkpointing under Unix". In: Usenix Winter 1995 Technical Conference, New Orleans, Jan 1995.

[6] PLANK, J. S. "An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on

Implementation and Performance". University of Tennessee, TR UT-CS-97-372., Jul. 1997.

[7] DUELL, J.; HARGROVE. P. & ROMAN, E. "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart". Berkeley Lab, TR LBNL-54941, 2003.

[8] ELNOZAHY, E. N.; JOHNSON, D. B. & WANG, Y. M. "A Survey of Rollback-Recovery Protocols in Message-Passing Systems". C. Mellon University, TR CMU-CS-96-181. 1996.

[9] MANIVANNAN, D.; NETZER, R. H. B.; SINGHAL M. "Finding Consistent Global Checkpoints in a Distributed Computation". In: IEEE Transactions on Parallel and Distributed Systems, vol. 8, n. 6, p. 623-627. 1997.

[10] NEVES, N. & FUCHS, W. K. "Coordinated Checkpointing without Direct Coordination". In: Proceedings of IEEE International Computer Performance & Dependability Symposium, pp. 23-31, Sep. 1998.

[11] VAIDYA, N. H. "Staggered Consistent Checkpointing". In: IEEE Transactions on Parallel and Distributed Systems. vol. 10, n. 7. p. 694-702. 1999.

[12] STELLNER, G. CoCheck: checkpointing and Process Migration for MPI. In: Proceedings of the 10th International Parallel Processing Consortium (IPPS 96). p. 526-531. 1996.

[13] CHEN, Y.; PLANK, J. S. & LI, K. "CLIP: A checkpointing tool for message-passing parallel programs". Princeton University, TR-543-97, May 1997.

[14] LITZKOW, M. et al. "Checkpointing and Migration of UNIX Processes in the Condor Distributed System". 1997. <www. cs.wisc.edu/condor/doc/ckpt97.ps>.

[15] BOSILCA, G.; et al. "MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes". In: Proceedings of IEEE SuperComputing 2002 (SC2002). Nov. 2002.

[16] LAM/MPI TEAM. "LAM/MPI Installation Guide version 7.1.1". Set. 2004. Available in <www.lam-mpi.org/ download/files/7.1.1-install.pdf>.

[17] LAM/MPI TEAM. "LAM/MPI User's Guide version 7.1.1". Set. 2004. Available in <www.lam-mpi.org/download/ files/7.1.1-user.pdf>.

[18] SQUYRES, J. M.; BARRET, B.; & LUMSDAINE, A. "Boot System Services Interface Modules for LAM/MPI". TR576, CS, Indiana University. Aug. 2003.

[19] SQUYRES, J. M.; BARRET, B.; & LUMSDAINE, A. "MPI Collective Operations System Services Interface Modules for LAM/MPI". TR577, CS, Indiana University. Aug. 2003.

[20] SQUYRES, J. M.; BARRET, B.; & LUMSDAINE, A. "Request Progression Interface System Services Interface Modules for LAM/MPI". TR579, CS, Indiana University. 2003.

[21] SANKARAN, S. et al. "Check-point-Restart Support System Services Interface (SSI) Modules for LAM/MPI". Technical Report TR578, CSD, Indiana University. 2003.

[22] MARTINS JR, Antonio da Silva; GONCALVES, Ronaldo A. L. Checkpointing Automático em Cluster MPI: Testes Preliminares. (In Portuguese). In: VI FITEM - FÓRUM DE INFORMÁTICA E TECNOLOGIA DE MARINGÁ, Maringá, Brazil, 2004.

[23] NEEDLEMAN, S. B. & WUNSCH, C. D. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". Journal of Molecular Biology, v. 48, pp. 443-453. 1970.

[24] SMITH, T. F. & WATERMAN, M. S. "Identification of common molecular subsequences". Journal of Molecular Biology, v. 147, pp. 195-197. 1981.

[25] GOTOH, O. "An improved algorithm for matching biological sequences". Journal of Molecular Biology, v. 162, pp. 705-708. 1982.

[26] ALVES, C. E. R.; CÁCERES, E. N.; DEHNE, F.; SONG, S. W. "A parallel wavefront algorithm for efficient biological sequence comparison". The 2003 International Conference on Computational Science and its Applications – ICCSA 2003. Lecture Notes in Computer Science, v. 2667. pp. 249-258. Berlim: Springer-Verlag. May, 2003.

[27] ALMEIDA Jr., N. F.; ALVES, C. E. R.; CÁCERES, E. N.; SONG, S. W. "Comparison of Genomes Using High-Performance Parallel Computing". 15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'03). 2003.

[28] ZHONG, H. & NIEH, J. "CRAK: Linux checkpointing / Restart as a kernel Module". Technical Report CUCS-014-01. Department of Computer Science. Columbia University, Nov. 2002.

[29] SUDAKOV, O. O. & MESHCHERYAKOV, E. S. "CHPOX: checkpointing for linuX". Sept. 2003. Available in <http://www.cluster.kiev.ua/tasks/chpx_eng.html>.

[30] MARTINS JR, Antonio da Silva; GONCALVES, Ronaldo A. L. "Extensões na LAM/MPI para Automatizar o Checkpoint e Tolerar Falhas em Cluster de Computadores". (In Portuguese). In: VI WSCAD (Workshop em Sistemas Computacionais de Alto Desempenho), Rio de Janeiro, Brazil, Oct., 2005.