Automatic Parallel Program Checkpointing in Message-Passing Environments

Andrey Smirnov
Faculty of Computational Mathematics
and Cybernetics
Moscow State University
Russia, 119992, Moscow, Leninskie Gory
Email: Smirnov.Andrey@gmail.com

***Abstract*—Problem of efficient cluster resources usage is very important, because of high demand for parallel computations. Checkpointing allows to manage cluster computing time more efficiently. In this article parallel programs checkpointing problems are discussed and implementation of automatic parallel checkpointing systems for MPI programs is presented. It is based on simple user-space portable checkpointing library with two different parallel program analysis approaches to checkpoint consistency.**

I. Introduction

Checkpoint[1] is saved program state that can be used to restore program execution from given point of time. The main goal of checkpoint usage is more efficient utilization of cluster computing time. The efficiency of cluster CPU time utilization may be increased with checkpoints in several ways: achieving higher transient fault tolerance and gaining more flexible task planning capabilities for job management system. In the first case checkpoints are created periodically, and are used to restore program execution in case of failure, thus minimizing CPU time loss. In the second case checkpointing capabilities allow job management system to suspend long-running jobs in favour of short-running ones: a checkpoint for long-running job is created, then long-running job is stopped and liberated cluster nodes are used to run short-running jobs; then long-running job is restored from its checkpoint and it continues its work. This scheduling policy improves cluster planning behaviour from user's point of view: short jobs are scheduled to run with shorter delays while execution time of long-running tasks is slightly increased.

Parallel program checkpointing raises problems irrelevant for sequential program checkpointing: individual processes of parallel program communicate through some kind of interconnection network using messages, and sum of states (checkpoints) of individual processes is not enough to represent whole parallel program state. Taking independent checkpoints of processes may lead to lost or duplicated messages during restore phase which is undesirable and may lead to incorrect computing results (or program crash).

Another question besides checkpoint correctness is efficiency of checkpointing service: checkpointing shouldn't cause overheads that could cancel its benefits. By checkpoint efficiency we mean overheads connected with creation of checkpoint and restore from checkpoint. Basically, checkpoint should contain collection of virtual memory occupied by all processes, so if each process uses a lot of memory and whole task has high degree of parallelism (a lot of processes in one task), approximate upper bound for checkpoint size could be found as multiplication of process count by average memory usage among processes. For large computational task this figure could be huge (several gigabytes or more), so the problem of minimizing checkpoint size (reducing overhead) is relevant.

[1]This work was supported by Russian Foundation for Basic Research grant number 05-01-00719-a

II. Theoretical aspects of checkpointing

All checkpointing methods by implementation techniques can be divided in two classes: user-level and system-level checkpoints. In the first case programmer builds checkpointing technique for any given program from ground, completely solving problem of saving and restoring program state. System-level checkpoints are implemented at operating system level or as a programming library; this implementation usually can be used for broad range of computational tasks and requires little or no modifications to program source. Implementation at operating system level (as a loadable kernel module) permits full state retention and restoration, though it could be less portable (it requires knowledge of internal OS structures) and requires administrator rights for deployment; implementation as a library (at user level) uses more portable interfaces and doesn't require administrator rights during deployment. Comparing user-level and system-level checkpoints we can say that user-level checkpoints can be more efficient, because they're designed specifically for one particular program; system-level checkpoints usually can't use advantages of one particular program, and so they aren't usually as effective as user-level checkpoints, but this approach greatly reduces checkpointing solution development efforts.

As it was said before, individual processes of parallel programs usually communicate by means of some interconnection network; in cluster systems it could be some kind of special network (like *Myrinet* or *SCI*) or ordinary network (like *Fast Ethernet*). This means that state of parallel program consists not only of processes' states, but also includes the

state of interconnection network. Commonly this network state (messages that were travelling through the network while checkpoint was taken) isn't saved, because saving and restoring this state correctly is a very complex task. And if we don't save this state, we should make some assumptions about network at the moment when checkpoint is taken. If these assumptions are not satisfied, we have a chance to face a lot of problems during restore from checkpoint.

A. Checkpoint consistency

Checkpoint of parallel program consists at least of checkpoints of individual processes. If we draw a timeline for parallel program, mark each sent and received message and mark checkpoint as individual points for all individual checkpoints of processes, every checkpoint may be called *consistent* or *inconsistent*: if we connect individual checkpoints with line, lines connecting send and receive slots for each message don't cross checkpoint line, then checkpoint is *consistent* (*strictly consistent*); all other checkpoints are *inconsistent*. In other words, checkpoint is consistent, if during restore from that checkpoint no message is lost and no message is duplicated.

There are two "classical" algorithms aimed at taking consistent checkpoints: *synchronous* and *asynchronous* checkpoints. In case of synchronous checkpoints, during the initialization of checkpointing procedure, auxiliary message "get ready for checkpoint" is sent to every process. If a process receives such message, it stops sending new messages, but continues to receive all messages. When all processes are sure that communication network is free of messages (for example, using "test" messages), every process creates its checkpoint and the execution of parallel program continues. This approach requires deep interaction between MPI implementation and checkpointing subsystem, but produced checkpoint is guaranteed to be consistent (by definition). Asynchronous checkpointing is simple: every individual process takes checkpoint whenever it feels comfortable. All such checkpoints are saved, and then some special algorithm used to find what individual checkpoints make up a consistent checkpoint for the whole parallel program (this algorithm should use some information about messages received and sent during program execution, message log for example). Such algorithm may be very complex, but there are other problems with this approach: one can imagine a condition when no set of asynchronous checkpoints make up consistent state (so-called *domino effect*).

Let's introduce another consistency-achievement approach: *preliminary synchronization*. In this approach, checkpoint-creation routine calls are inserted into parallel program source code in such places, that produced checkpoint is guaranteed to be consistent. Places for checkpoint creation are chosen based on following criteria:

- communication channels are empty;
- no process is blocked in receive message state.

We will call these assertions *preliminary consistent checkpointing criteria*. Obviously these criteria guarantee consistency of produced checkpoint, indeed: no message will be lost (first condition) and all processes are ready for checkpoint creation (second condition). Points in program execution conforming to these criteria are usually located at end of iteration for iterative MPI programs, after collective MPI operations affecting all processes. This approach offers small synchronization times to take a checkpoint with a price of some preliminary work before program execution.

Places in program conforming to these criteria may be chosen in several ways:

- manual choice by programmer (may require some insight into message communication pattern, it could be rather hard to prove that chosen place conforms to criteria);
- static analysis of the program (source code of the program is analyzed);
- dynamic analysis of the program (message traces for several test runs of the program are analyzed).

B. Checkpointing efficiency

By checkpointing efficiency achievement we mean reducing checkpoint creation and restore cost. The biggest part of this cost comes from input-output operations, required to write (read) checkpoint image to some storage, in fact checkpoint size can be as large as sum of all parallel program processes memory requirements (sum of all processes' data sizes). So, one could expect checkpointing efficiency achievement if input-output costs are decreased or if checkpoint's size is reduced. In the following paragraphs we will give a short overview of different methods for efficiency achievement.

1) Data compression: Since checkpoint consists of program data, and usually program data could be easily compressed (with assumption of data redundancy), use of compression algorithm to compress checkpoint data can significantly reduce its size. And, as the main goal is to reduce checkpointing costs, algorithm should be chosen as a trade-off between compression ratio and compression/decompression speed. Such algorithm can be found among dictionary compression algorithms family. If we know in advance program data pattern, dedicated compression algorithm could be used to achieve better results.

2) Asynchronous checkpoints: Usually, checkpoint creation process interrupts execution of the program until the checkpoint creation finishes. But checkpoint creation consists basically of input-output operations, and program execution is CPU-bound, so checkpoint could be taken in parallel with program execution without much loss in productivity. Since the checkpoint should contain program state at the moment of checkpoint creation, there should be some convenient way to save that state. In UNIX operating systems this can be done with `fork()` system call.

3) Exclusion of unused memory: There are some program data areas which are rewritten with first access after checkpoint creation (program execution after checkpoint doesn't depend on data in such areas). So, we shouldn't save such parts of data in checkpoint, since its content isn't important to program execution after restore from checkpoint. Examples of such data areas are temporary arrays which are used during

computational iteration if checkpoint is taken at the end of iteration.

III. Checkpointing implementations survey

There are a lot of single-process checkpointing implementations, we could mention some of them: [1], [2]. These tools can't be used for parallel program checkpointing as such, but are suitable as a basis for parallel checkpointing solution.

There are projects that provide some tools for user-level checkpointing, but they can only be used to cut down checkpointing development effort a little, and programmer should do all the work associated with program state save and restore. Also there are several highly specialised checkpointing solutions like [3], proprietary checkpointing solution used in IBM high-performance systems (with proprietary MPI implementation), but they are not considered here.

Complete working solution [4] is actively developed for LAM/MPI library built on the basis of Berkley Labs Checkpoint/Restart System. This system performs deep checkpointing of communication layer: every communication driver provides hooks for flushing all its unsent messages, so it is very similar to synchronous consistency checkpointing algorithm.

More complete survey of parallel checkpointing solutions can be found in [5].

Our experimental implementation is oriented at MPICH MPI interface library, which is more widely used in cluster environments (than LAM/MPI). It consists of two parts:

1) basic checkpointing library `libcheckpoint`, that checkpoints single process, saves program and operating system state, correctly restores program execution from saved checkpoint; the library also correctly checkpoints and restarts parallel MPI program execution given that checkpointing is performed at states that meet preliminary consistency checkpointing criteria (see II-A);
2) two solutions eliminating the requirement of manual search for places in program meeting these criteria:
   a) MPI events trace analysis resulting in optimal checkpoint placement aids (static program analysis);
   b) online distributed algorithm that finds such places during program execution (dynamic program analysis).

Our solution is highly portable, there is no need for deep integration into communication driver in MPI implementation, it doesn't require changes to MPICH and base system libraries. Parallel program analysis methods don't depend on any MPI implementation, they are based only on MPI Standard.

IV. Basic checkpointing implementation: libcheckpoint

MPICH library was chosen as the most popular MPI implementation. Several communication networks (supported by MPICH) were chosen for the first version of `libcheckpoint` library:

- shared memory (pure SMP-systems);
- Myrinet network (high-performance clusters);
- Ethernet.

Preliminary synchronization was chosen as a method for ensuring checkpoint consistency. This method gives low synchronization costs, requires storing of only one set of checkpoints for each program and doesn't lead to message pattern changes (doesn't require service messages). Three most useful efficiency achievement methods were chosen: data compression, asynchronous checkpointing and manual unused memory exclusion.

During the design and development there were several main goals:

- portability;
- no changes to system libraries (or MPICH library);
- specific architecture tuning during the installation;
- maximum efficiency while retaining checkpoint correctness.

Libcheckpoint library essential features are:

- Automatic checkpoints at system-level; `libcheckpoint` is implemented in C language as programming library.
- Several architectures are supported: FreeBSD/x86, FreeBSD/amd64, Linux/x86, Linux/Alpha.
- Supported programming languages:C/C++, Fortran.
- Process state save/restore (data & stack segments, heap, process context).
- Operating system state save/restore (memory mapped regions, open files, signal handlers).
- Process migration between machines with identical architectures.
- Library efficiency evaluation subsystem.

A. Results of efficiency evaluation

In order to test efficiency of the library, computational task solving some hydrodynamics problem was chosen. Tests were taken on Russian high-performance cluster, MVS-1000M. The task was modified to take 10 checkpoints while it was running (total task running time was about 50 minutes). Average checkpoint creation times (among 10 checkpoints) were calculated, and several tests were run with different number of parallel processes in each run.

On average, it took 9,35 seconds to take one checkpoint. If checkpoints are created every 30 minutes, this costs 0,5% of loss in time. Average restore time was 15 seconds, and this seems acceptable.

V. Automatic checkpoint placement

This section describes two proposed and implemented solutions that can be used in conjunction with `libcheckpoint` in order to find places in parallel program that meet preliminary checkpoint consistency criteria (see II-A).

A. MPI trace analysis

The basic scheme of this approach can be described as follows: parallel program is started with MPI trace library compiled in, MPI event trace is produced, it is fed into

trace analyzer that outputs proposed checkpoint placements, checkpointing calls are integrated into source code of program and it can now be started with checkpointing enabled.

MPI trace consists only of two types of events: message send and message receive. Other events (like internal computing) aren't important for checkpoint placement, and other MPI operations can be simulated with these events provided that cause-effect dependencies are the same.

Relation of partial order is introduced on set of such events in natural way:

1) $e_1 \prec e_2$ if they happened in the same process, e_1 earlier than e_2;
2) $e_1 \prec e_2$ if e_1 is corresponding send operation to e_2 - receive operation;
3) $e_1 \prec e_2$ if there exists sequence of events $\hat{e_1}, \hat{e_2}, \dots \hat{e_n}$ so that $e_1 \prec \hat{e_1} \prec \hat{e_2} \prec \dots \prec \hat{e_n} \prec e_2$ in sense of points 1 and 2.

In order to discover such dependency in parallel program trace, vector timers are used (vector times are described in detail in [6]). Vector timers are extension to Lamport's clock [7]. Vector timer relation and partial cause-effect dependency for two events are equivalent, so MPI tracer collects vector timers for all events and collected timer values are used to discover event cause-effect relations during analysis phase.

Cause-effect event relations are used to analyze non-deterministic MPI programs and to define necessary condition of consistency: let a_i be the event right before the checkpoint in process i and b_i the event right after checkpoint in the same process, then $\forall k, l \in \{1 \dots n\} : a_l \not\succ b_k$ should be true for all consistent checkpoint placements. This condition means that event after checkpoint can't depend on event before the checkpoint. The sufficient condition is simple: all send-receive operations are finished at the time of checkpoint, so every send event has its corresponding receive event completed and vice versa.

The algorithm was developed to find all places in program meeting the sufficient condition. Algorithm searches through set of events considering the set of pairs $\{(e_1, \hat{e_1}), (e_2, \hat{e_2}), \dots, (e_n, \hat{e_n})\}$ as a potential checkpoint placement candidate, where n - number of processes in parallel program and $\hat{e_i}$ is event immediately after e_i in process i. Each such set is checked using sufficient condition of consistency. During the search necessary consistency condition and several other propositions are used to cut down the depth of search. The question of NP-completeness of this problem stays unanswered at the moment.

MPI tracing library was implemented in C language, it intercepts calls to MPI functions, storing the trace of events happened in parallel program with information about source code location of such events (in order to put checkpoint creation calls into program's source code). With each send or receive operation the library sends its local vector timer and performs adjust operations on local timer when it receives timers from other processes. Timers are sent and received using separate calls to `MPI_Send` and `MPI_Receive` in such fashion that these additional operations don't modify original program message passing semantics. Algorithm for consistent checkpoint placement was implemented in Python language with tests and simple trace visualizer.

This approach has one disadvantage: checkpoint placement depends on MPI program trace. We need to perform first run of the program to get such trace, and this run goes without checkpointing enabled. Ideal first run should be the identical to all subsequent, but usually we don't have to run the task with the same input several times. So, first run could be modified in such a way that it requires less computing power but retains the same message-passing pattern. For example, real calculations could be taken out of the program or for iterative algorithms number of iterations could be cut down significantly if all iterations are identical in terms of message exchange. Other part of this disadvantage is potential dependency of program trace on its input data, so that checkpoint placement discovered for one input is useless for other inputs. In such situations this approach can't be applied.

B. Online distributed algorithm

This approach is based on distributed algorithm that runs in "parallel" with user program and creates checkpoint at nearest consistent point on program request. Algorithm intercepts calls to MPI functions and stores locally (in process) information about number of messages sent and received to each other process.

Algorithm has two phases: inactive and active. Actions performed in inactive phase are:

1) Let i be the current process number, and n be the count of processes in parallel program.
2) Each process i stores two vectors of length n: s^i and r^i, at program start they are initialized with zeroes: $s^i = [0, 0, \dots, 0]$ and $r^i = [0, 0, \dots, 0]^T$
3) Before sending the message to process j process i increments corresponding element of vector s^i: $s^i_j = s^i_j + 1$, and after the messages is received from process j corresponding element of vector r^i is incremented: $r^i_j = r^i_j + 1$.
4) If algorithm gets signal "it's time to create a checkpoint" from program, it goes to active phase, and sends internal message "go to active phase" to all other processes.
5) If algorithm receives internal message "go to active phase", it goes to active phase.

In active phase algorithm performs following actions:

1) Processes exchange their local vectors s^i and r^i so that each process builds complete view of processes' state: matrices $S = \begin{bmatrix} s^1 \\ s^2 \\ \dots \\ s^n \end{bmatrix} = \{s_{kl}\}$ and $R = \begin{bmatrix} r^1 & r^2 & \dots & r^n \end{bmatrix} = \{r_{kl}\}$, where s_{kl} is number of messages that process k sent to process l and r_{kl} is number of messages that process l received from process k.
2) Each process checks the condition $\forall k, l = 1 \dots n : s_{kl} = r_{kl}$, if it becomes true, each process creates its

local checkpoint, algorithm goes to inactive phase, and program execution continues.

3) If for the current process (with number i) $\exists j : s_{ji} < r_{ji}$ process continues its normal execution until it sends or receives message from other process or internal algorithm message is received (go to step 4 or 5). This condition means that message was sent from process j to current process i, but was not yet received. If $\forall j : s_{ji} \geq r_{ji}$ execution is suspended until internal messages is received (go to step 5). In this case inconsistency exists, but it isn't caused by current process.
4) If the program sends or receives message (not internal algorithm message), vectors r^i and s^i are updated (as in inactive phase), matrices S and R are corrected locally and updates to matrices are distributed via internal messages to other processes. Algorithm execution continues from step 2.
5) If the process receives internal message about S and R matrices being updated, it performs the updates and execution continues from step 2.

This algorithm version is for MPI programs that send messages with one fixed tag, for multiple tags algorithm is more complex, it provides copies of S and R matrices for all used tags. If condition of step 2 is met, it is evident that the current program state is consistent according to preliminary consistent checkpointing criteria (see II-A). It could be proved that this condition is met at the same moment in all processes or isn't met at all.

At the time of checkpoint request from the program the algorithm suspends execution of all processes, calculates current inconsistencies, and performs decisions locally in each process leading to some processes continuing execution if its execution could resolve these inconsistencies (messages already sent but not received yet). So the search is performed to find such consistent point. This search could lead to other inconsistencies which in turn lead to processes continuing execution. But, if such consistent point exists, it will be found, if it doesn't exist, this program doesn't have points suitable for our criteria, and our approach can't be applied here. But such programs in practice are too rare.

Algorithm was implemented in C language as a library integrated with the `libcheckpoint` checkpointing library. This implementation turns all blocking MPI operations into non-blocking, allowing algorithm messages to co-exist with normal program message exchange. Algorithm gets control on every MPI call performed by program and, depending on the current phase and performs actions described above.

VI. Conclusion

Library `libcheckpoint` in conjunction with one of the approaches for automatic checkpoint placement can be used to produce automatic checkpointing system for parallel programs using MPICH library running at clusters with different communication protocols. Choice between two checkpoint placement approaches is based on the nature of the task, its non-determinism, dependency on input data, running time and so on. Presented automatic parallel program checkpointing system can be used for fault-tolerance and efficient task management, that helps to manage cluster computing time more efficiently.

Acknowledgment

The author would like to thank his scientific adviser, Dr. Igor V. Mashechkin (Moscow State University).

References

[1] J. Duell, "The design and implementation of berkeley lab's linux checkpoint/restart," Lawrence Berkeley National Laboratory, Tech. Rep., Apr. 2002, publicateion LBNL-54941. [Online]. Available: http://repositories.cdlib.org/lbnl/LBNL-54941/

[2] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under Unix," in *Proceedings of USENIX Winter1995 Technical Conference*, New Orleans, Louisiana/U.S.A., Jan. 1995, pp. 213–224.

[3] A. J. Ferrari, S. J. Chapin, and A. S. Grimshaw, "Process introspection: A heterogeneous checkpoint/restart mechanism based on automatic code modification," Tech. Rep. CS-97-05, 25, 1997.

[4] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The Lam/Mpi Checkpoint/Restart Framework: System-Initiated Checkpointing," *International Journal of High Performance Computing Applications*, vol. 19, no. 4, pp. 479–493, 2005.

[5] E. Roman, "A survey of checkpoint/restart implementations," Lawrence Berkeley National Laboratory, Tech. Rep., 2002, paper LBNL-54942. [Online]. Available: http://ftg.lbl.gov/CheckpointRestart/checkpointSurvey-020724b.pdf

[6] R. H. B. Netzer and B. P. Miller, "Optimal tracing and replay for debugging message-passing parallel programs," Tech. Rep. CS-94-32, 1994. [Online]. Available: citeseer.ist.psu.edu/netzer92optimal.html

[7] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–564, 1978.