Scalable Group-based Checkpoint/Restart for Large-Scale Message-passing Systems[1]

Justin C. Y. Ho, Cho-Li Wang and Francis C. M. Lau
Department of Computer Science,
The University of Hong Kong,
{cyho2,clwang,fcmlau}@cs.hku.hk

Abstract

The ever increasing number of processors used in parallel computers is making fault tolerance support in large-scale parallel systems more and more important. We discuss the inadequacies of existing system-level checkpointing solutions for message-passing applications as the system scales up. We analyze the coordination cost and blocking behavior of two current MPI implementations with checkpointing support. A group-based solution combining coordinated checkpointing and message logging is then proposed. Experiment results demonstrate its better performance and scalability than LAM/MPI and MPICH-VCL. To assist group formation, a method to analyze the communication behaviors of the application is proposed.

1. Introduction

Over the past few years, there has been a rapid growth in system size in terms of number of processors. According to the list of TOP500 Supercomputer Sites, the share of systems comprising more than 512 processors jumped from 83% in June 2007 to 97.8% in November 2007. Back in the year 2003, only 23.4% of the systems were of such scale. Increasing system scale translates naturally into increasing computing power, but on the other hand the system becomes more vulnerable to failures. Fault tolerance support becomes more important than before.

Among the many methods to achieve fault tolerance, checkpointing is one of the most well-established. With this method, checkpoints of applications at certain time instants are written to stable storage; later on, if and when a system failure occurs, the checkpoint images are retrieved to restart the affected processes, which prevents complete loss of computation work.

The increase in system size brings new challenges to checkpointing and restarting a typical message-passing application as the system size would cause more control and coordination overheads.

Elnozahy et al. [6] gave a detailed survey of various checkpointing strategies. Among them, coordinated checkpointing usually incurs high coordination cost as checkpoints and rollbacks have to be done globally. Moreover, recovery by a global restart, which involves all other non-failed processes, would lose all the useful work done by these normal processes. Assuming that failures only occur in a small region of a large system, such work losses would be wasteful. To reduce such losses, checkpointing could be done more frequently. However, frequent checkpoints may slowdown the whole system, resulting in a worse effect than the failures [13].

Non-blocking coordinated checkpointing may be used to reduce the coordination costs. However, when checkpoint is in progress, there might be a short period of time when the processes are not allowed to send any messages. If other processes are blocked waiting for such messages, their execution will be paused. The delay may propagate to other processes, and eventually the whole application. Therefore non-blocking checkpoints may actually become blocking, defeating its purpose. The problem gets worse in larger systems as there would be more messages and message dependencies would become complex.

For checkpointing approaches that require additional operations for handling messages, such as uncoordinated checkpointing, message logging [2] and communication induced checkpointing (CIC) [11], their performance would degrade when the system size scales up, due to the increased overheads in message logging and retransmissions. For example, CIC-based

[1] This research is supported by a Hong Kong RGC grant (HKU 7176/06E) and a China 863 grant (2006AA01A111).

systems have to deal with the increased number of message transfers, either to detect undesirable patterns such as Z-cycles [11], or to piggyback information on messages to assist decision making. Other studies also showed that CIC-based solutions are unfavorable in large-scale systems [1].

Similarly for message logging approaches, overhead is induced in logging each message transfer. The effect could be very prominent in pessimistic logging; in causal logging [15], it is difficult to track the message dependencies.

Uncoordinated checkpointing also involves message handling overheads. Even worse, it suffers from the high probability of having the *domino effect* while forming a consistent *recovery line*. Ohara et al. in [12] illustrated the low probability of having a recovery line from uncoordinated checkpoints of large systems.

We set out to develop a more scalable and efficient checkpoint/restart solution for large-scale systems. We believe a good solution should possess the following characteristics: (1) low coordination and control overheads via non-global checkpoints; (2) low message management overheads in supporting non-global checkpoints and restarts; and (3) flexibility in real-life usage. Many existing checkpointing approaches failed to meet at least one of the above requirements, which could lead to poor performance when the system size is large, rendering the approach impractical.

In this paper, we present a group-based checkpointing system. To facilitate process group formation, a light-weight MPI communication tracer is proposed. The trace outputs are used to identify those intensively communicating MPI processes which will form a group. By allowing a group of processes as a basic unit to checkpointing, global checkpoints and restarts can be avoided. Within a group, checkpoints are done in a coordinated manner. Consistency among the groups is maintained by keeping logs of intergroup messages. Under this approach, the checkpoint coordination overhead is reduced through non-global checkpoints; and only a smaller number of messages are logged, thus reducing the overhead in message transfers. We implemented the group-based checkpoint system based on LAM/MPI [14]. Experiment results with HPL [16] and NPB [17] show that the group-based solution can reduce checkpoint overhead by over 80% when compared with LAM/MPI which uses global, coordinated checkpointing.

The remainder of the paper is organized as follows. Section 2 describes the limitations of existing works. Section 3 presents the proposed solution, whose implementation details are in Section 4. Experiment results and performance comparisons against existing works are presented in Section 5. Several related works are briefly discussed in Section 6. Finally, Section 7 concludes the paper.

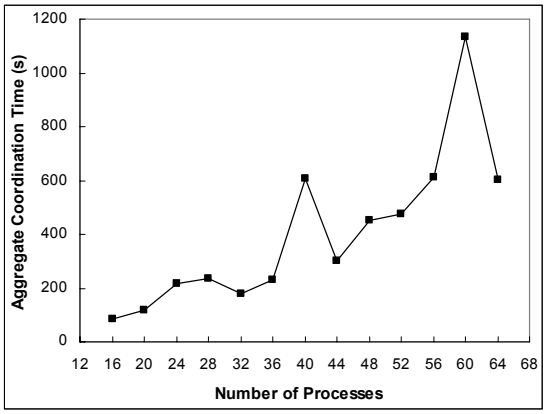


Figure 1 Checkpoint coordination time in HPL with LAM/MPI

2. Limitations of Existing Works

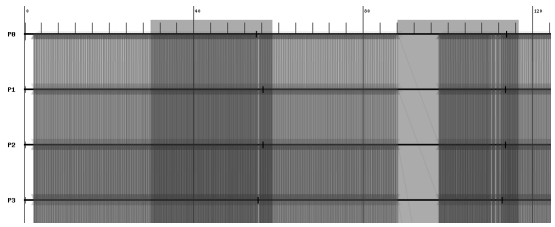
In this section we present our findings on the limitations of two current MPI implementations with checkpointing support: LAM/MPI and MPICH-VCL [4].

2.1. Coordination cost in LAM/MPI

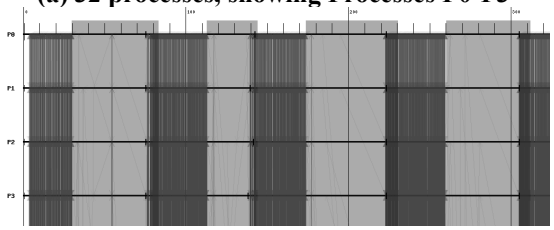
Figure 1 shows the sum of time spent by all processes in coordinating one global checkpoint, when running HPL with LAM/MPI. Coordination time is estimated by excluding the time spent in creating the actual checkpoint image. By timing each step of the checkpoint process, it is found that in LAM/MPI, the time spent in coordination work varies largely from almost instantly up to a few seconds. Since the application can not make any progress during the checkpoint, this time is wasted. Besides the gradual increase in duration, any unexpected delay in the processes may greatly affect the overall performance, as shown in the Figure when the number of processes is equal to 40 and 60 respectively.

2.2. Blocking behavior in MPICH-VCL

MPICH-VCL, which follows Chandy and Lamport's non-blocking coordinated checkpointing algorithm [3], exhibits blocking behavior when the system is scaled up. The CG application from the NPB



(a) 32 processes, showing Processes P0-P3



(b) 128 processes, showing Processes P0-P3

Figure 2 MPI Trace diagram for CG using MPICH-VCL, with checkpoints every 30s

Algorithm 1 Group-based checkpoint/restart

Definitions:
- R_X: volume of messages in bytes received from Process X
- RR_X: Recorded value of R_X before the latest checkpoint
- S_X: volume of messages in bytes sent to Process X
- V_P: piggybacked value of RR_X from Process P

At process start:
- Read group definitions; Identify the process's own group and the group members

On sending a message to process *P*:
- If P is not within the group: Log message asynchronously. If this is the first message sending to P after a checkpoint, piggyback RR_P to the message
- Update S_P

On receiving a message from process *P*:
- Update R_P
- If there is a piggybacked value V_P: Do garbage collection of message logs for P according to V_P

On receiving a group checkpoint request:
- Synchronize message logs
- For each of the out-of-group process Q, remember R_Q as RR_Q
- Coordinate with other group members to create a consistent checkpoint of the group
- Wait until all group members finish taking the checkpoint

On restart:
- For each of the out-of-group process Q, request Q to replay messages and determine amount of messages to skip sending to Q by exchanging R_X and S_X values
- Wait until all group members finish preparing the restart

suite exhibits non-stop message transfers throughout the execution. In other words, the application can not progress when there is no message. Figure 2 shows partial MPI traces of CG, running with 32 and 128 processes using MPICH-VCL, with checkpoints taking place every 30 seconds. Messages are shown as arrows from the source to the destination process. The checkpoint duration is shown as light grey blocks overlaid on the diagram.

Figure 2a shows the trace with 32 processes. There are portions in the light grey blocks that appear darker because there are message transfers during the period, i.e., the application was able to make progress. This is due to the non-blocking checkpoint process. However, light grey "gaps" in Figure 2b reveal that the execution was actually paused when checkpointing 128 processes.

The problem gets worse when the system scales, as shown in Figure 2b where 128 processing nodes were used. Note that with 128 nodes the "gaps" nearly span the whole checkpoint process in every checkpoint, which is very different from the relatively smaller or even the lack of "gaps" in the case of 32 nodes. The case for 128 nodes also shows that the checkpoint process actually spent away more than 50% of the total execution time. Such overhead is unbearable.

3. Group-based Checkpoint/Restart

3.1. Design

The proposed group-based solution combines coordinated checkpointing and message logging. The amount of work done on messages is reduced by grouping processes that communicate frequently. At the same time, coordinated checkpointing within a group tends to require less time in coordination than if it is done globally.

Algorithm 1 describes the scheme of the group-based checkpoint/restart. Checkpoints are coordinated within each group. Among different groups there is no coordination. Only inter-group messages are logged by the sender asynchronously, and intra-group messages need not to be logged. Message logs are flushed to storage right before a checkpoint. Therefore each successful checkpoint comes with a correct set of message logs. When a message is sent to a certain process, the volume of messages sent to that process is recorded in terms of bytes. Volumes of messages received are recorded similarly. This is to determine the volume of messages to replay or skip during a restart. For the first message sending to any

process after a checkpoint, the volume of messages *received* from that process before the checkpoint is piggybacked onto the message, such that garbage collection of message logs can be done accordingly.

Under this approach, checkpoints and restarts can be done in units of groups. Coordination cost is reduced as there are fewer processes participating in a coordinated checkpoint. The amount of messages required to be logged is also reduced comparing with pure message logging approaches. Figure 3 illustrates a comparison of group-based checkpoint against a typical coordinated checkpoint and a typical message logging approach. In the next section, we discuss how to partition processes into groups and how to select the group members according to the inter-process communication patterns.

3.2. Trace assisted group formation

A good suggestion for group formation can come from analyzing MPI communication traces as illustrated in Algorithm 2. *Send* records in the trace are extracted, which have the format of *(source, destination, size)*. Next, those records having the same unordered source/destination pair are found, the total number of messages and the total message size are calculated, and they are stored as a list of tuples of the form *(process ranks, count, size)*. As we would like to give higher priority to process groups that communicate more, the list is sorted by *size*, then by *count* in descending order. Each tuple in this *input list* is extracted, and its two process ranks are searched in the tuples in the *output list*, to check if any merging with the existing groups can be done. Merging groups requires that they have at least one common process. Groups will not be merged if it would exceed the maximum group size defined. If no such group can be found, the tuple is inserted to the *output list*, representing a new group of two processes. The merging continues until the *input list* is exhausted. The resultant groups may not be of equal sizes and may not reach the given maximum group size. This is normal behavior since unrelated groups without any message transfers should not be merged into a group.

We set an upper bound on the group size to enforce process grouping and avoid global checkpoint coordination in the first place. The default value is the square root of the number of processors. Indeed, the parameter can be adjusted according to the hardware environment. For example, when high speed networks are used, a larger maximum group size may be chosen

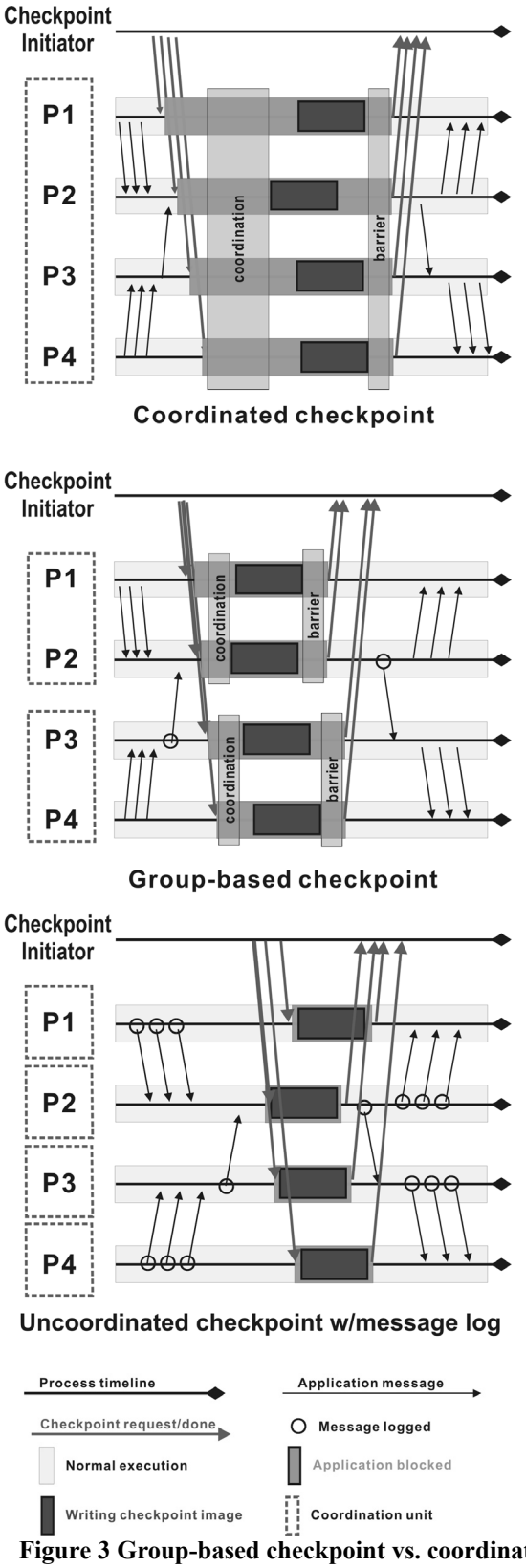


Figure 3 Group-based checkpoint vs. coordinated checkpoint vs. message logging

Algorithm 2 Group Formation

Algorithm inputs:
- send operations from the MPI trace, identified by *(SRC, DST,Z),* where *SRC, DST* and *Z* stand for source, destination and message size respectively
- maximum group size *G*
- total number of processes *n*

Define data structures:
- Tuples L=(P, N, S), where P consists of processes to be in a group, N is the total number of the messages, and S is the total size of the N messages. Each tuple represents a set of processes that communicates.
- *L,M,T* as lists of tuples

Define operations on tuples $L_i = (P_i, N_i, S_i)$, list *M*:
- $L_i + L_j = (P_i \cup P_j, N_i + N_j, S_i + S_j)$
- *numP*(L_i): returns number of processes in P_i
- *insert*(L_i, *M*): insert L_i to *M*
- *find*(*P, M*): returns the reference of the first tuple $R=(P_R, N_R, S_R)$ in *M* that *P* belongs to P_R
- *delete*(M_i *M*): remove M_i from *M*
- *length*(*M*): returns number of tuples in *M*

Preprocessing:
- For each trace record t_i = *(SRC, DST, Z)*: Create tuple $T_i = (SRC \cup DST, 1, Z)$; *insert*(T_i, *T*)
- For all tuples $T_0, T_1, \ldots, T_i$ in *T* having the same set of process ranks: *insert*($T_0+T_1+\ldots+T_i$, *L*) ; delete $T_0, T_1, \ldots, T_i$ from *T* ; repeat until *T* is empty
- Sort *L* descendingly by *S*, then by *N*, finally by *P*
- Note that each tuple L_i in *L* contains exactly 2 processes. Therefore L_i can be represented by $(P_1 \cup P_2, N, S)$. Let there are *l* tuples in *L*

Main loop: For each tuple $L_i = (P_1 \cup P_2, N, S)$ in *L*, *i=0,1,...l*
- $R_1 \leftarrow$ *find*(P_1, *M*) ; $R_2 \leftarrow$ *find*(P_2, *M*)
- if R_1=NULL & R_2=NULL: *insert*(L_i, *M*)
- if only R_2 is NULL & *numP*($R_1 + L_i$) $\leq G$: $R_1 \leftarrow R_1 + L_i$
- if only R_1 is NULL & *numP*($R_2 + L_i$) $\leq G$: $R_2 \leftarrow R_2 + L_i$
- if both R_1 and R_2 is not NULL & $R_1 = R_2$: $R_1 \leftarrow R_1 + L_i$
- if both R_1 and R_2 is not NULL & $R_1 \neq R_2$ & *numP*($R_1 + R_2$) $\leq G$: $R_1 \leftarrow R_1 + R_2 + L_i$; *delete*(R_2,*M*)

Algorithm output:
- *M. P*rocess ranks from each tuple of *M* form a group

to reduce the amount of message logs, so that overheads due to message latency can be reduced to retain the benefits of using a high speed network. It is also possible to coordinate checkpoints of a larger group efficiently using a faster network. In slower networks, having large groups may not be a good approach as there would be more in-transit messages to be cleared than having a smaller group, and synchronization work would be less efficient.

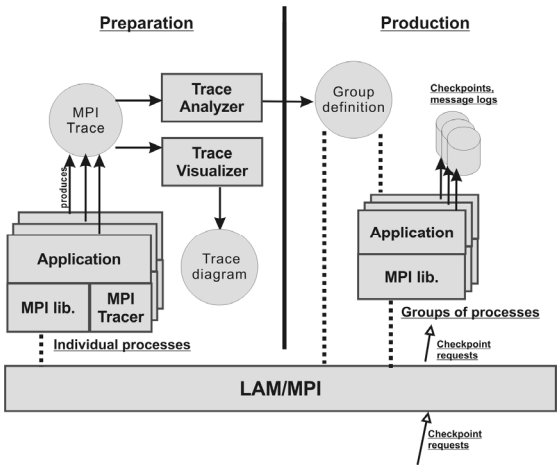


Figure 4 System diagram and workflow

4. Implementation

Figure 4 presents a system diagram and describes the workflow to perform group-based checkpointing. To prepare a group definition, the MPI tracer library is linked with the application, and is executed to prepare a set of MPI trace. The trace can be visualized as a diagram, or to be analyzed to produce a *group definition file*. Subsequent executions may then make use of the same group definition file, and the MPI tracer library would not be needed anymore. `mpirun` is responsible to receive checkpoint requests from the system or the user, and to propagate the requests to MPI processes.

Implementation of group-based checkpoint is done on top of LAM/MPI by enhancing its original `CRTCP`, `CRLAM` and `CRMPI` SSI modules which provide most of the checkpointing functionalities [14]. The `mpirun` utility is also modified to read a *checkpoint target file* which specifies which group(s) to checkpoint, and spawn one child for each group to propagate the checkpoint requests. All coordination work and barriers are limited within the group. One key step in the coordination work is to receive all pending messages from the group members. When a group finishes its checkpoint, it resumes normal execution regardless of other groups' progress in their checkpoint. After all the groups have finished their checkpoint, `mpirun` finally checkpoints itself. During a restart, after re-creating process spaces and updating LAM/MPI internal structures, each pair of out-of-group processes exchange the volumes of messages sent/received, and messages are then replayed or skipped accordingly. After this step all processes return to their normal execution.

5. Experiment Results

The system used in the following experiments is the HKU Gideon 300 Cluster. Each of the computing nodes is equipped with single Pentium 4 2.0 GHz processor, 512MB of physical memory, connected with Fast Ethernet networks. The computing nodes run on Linux kernel version 2.4.22. Up to 128 nodes were used in the experiments, where each node executes at most one MPI process. Each experiment was repeated five times to obtain an averaged result. LAM/MPI version 7.1.3b is used to implement the group-based checkpoint system, with BLCR [5] version 0.4.2 as the underlying system level checkpointer. Unless otherwise stated, checkpoint images and message logs are stored on the local hard disk. Experiments were carried out on two applications: High Performance Linpack (HPL) version 1.0a and NAS Parallel Benchmarks (NPB) version 2.4.

5.1. Experiments with HPL

The experiments with HPL are done with problem size (N) 20000 and block size (NB) 120. 16 to 128 processes are used, in 8-process increments. One process grid (P×Q) setting is chosen for each scale. P is fixed at 8 for the experiments. Process mapping is in row-major order.

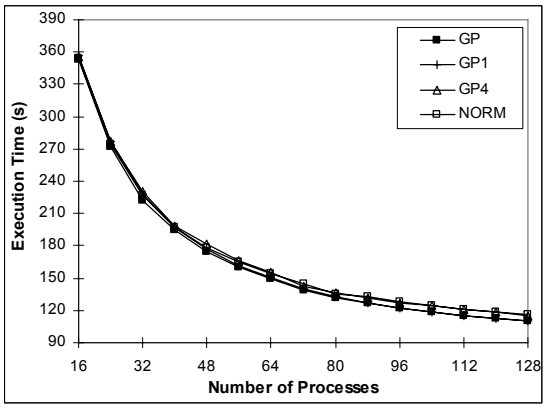
The MPI trace analysis results matched with the P×Q setting of the executions, that there would be Q groups with size P, with the process ranks in a round-robin fashion. This is a good example that the group formations can match with the application behavior. Table 1 shows an example for 32 processes (8×4). At 60 seconds after starting the program, all groups are signaled to take one checkpoint. After the program finishes it is immediately restarted from the only checkpoint, to measure the time needed to resume normal operations.

Three grouping methods are used and compared with the original LAM/MPI coordinated checkpoint, and we use the following notations:

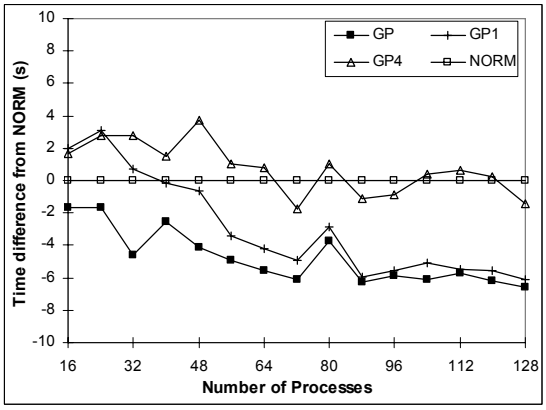
- GP: Group formation decided by obtaining and analyzing MPI traces
- GP1: One process per group, i.e., uncoordinated checkpoint with message log
- GP4: Four groups with sequential process ranks; which represents an ad-hoc group formation
- NORM: One group only, equivalent to the original LAM/MPI global coordinated checkpoint

Group # (Q groups)	Process ranks (P processes)
1	0, 4, 8, 12, 16, 20, 24, 28
2	1, 5, 9, 13, 17, 21, 25, 29
3	2, 6, 10, 14, 18, 22, 26, 30
4	3, 7, 11, 15, 19, 23, 27, 31

Table 1 Group formation for HPL, 32 processes (P×Q=8×4)



(a) Execution time

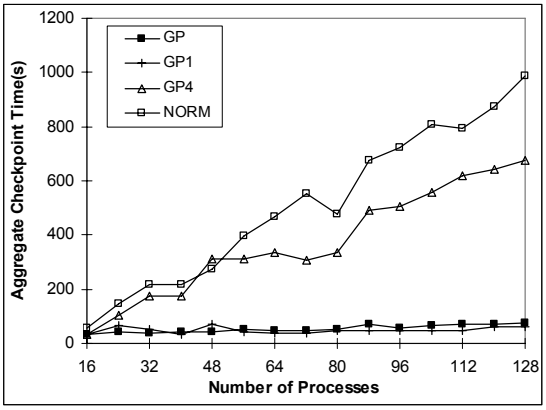


(b) Difference from NORM (lower is better)

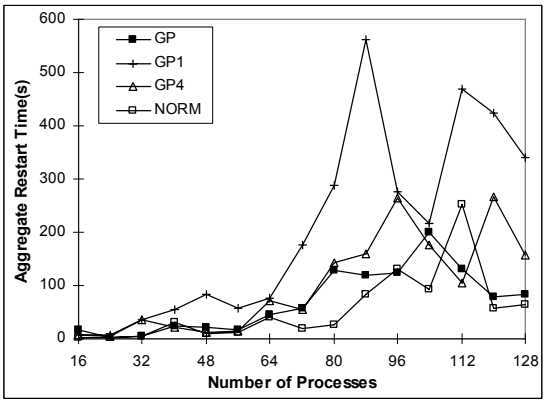
Figure 5 Execution time with one checkpoint at t=60s

Checkpoint time is measured per process, from the receipt of checkpoint signal until the process resumes normal execution. After all the processes finish the checkpoint, the `mpirun` process would do a checkpoint for itself. This period is not timed because it does not affect the normal execution anymore. Restart time is measured similarly, from the recreation of the process to its return to normal execution. Figure 5 to Figure 10 show the experiment results with HPL.

Figure 5a shows the execution time of HPL from start to finish, with one checkpoint at the 60th second of the execution. Despite the adverse effect of message logging with group-based checkpoints in GP, GP1 and



(a) Checkpoint



(b) Restart

Figure 6 Summed checkpoint and restart time

GP4, they perform as well as NORM. The graph for NORM is fluctuating, indicating that there had been some variable delay in the checkpoint process and the delay is reflected in the total execution time. This did not happen in GP, GP1 and GP4. For comparison, Figure 5b shows their difference in the time with respect to NORM. GP actually outperforms NORM, because the saving in checkpoint time is more than enough to compensate for the slowdown due to logging messages. The performance edge in GP over NORM steadily increases when the system is scaled up, indicating that GP is more scalable than NORM.

Figure 6 shows the sum of time spent in every process during checkpoint and restart. This represents the total CPU time of the system spent in such operations. Figure 6a shows checkpoint time. As expected, GP1 performed the best since there was absolutely no coordination. GP, which represents a good grouping method, showed similar performance to GP1. This is because in GP processes that work closely are grouped together, and the processes would have similar progress. Therefore, when coordinating the processes, waiting time is minimal and there are fewer

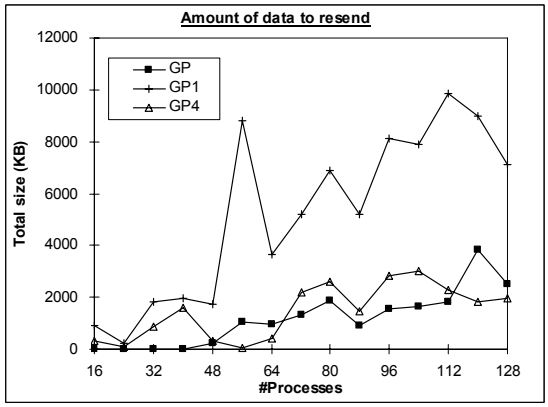


Figure 7 Amount of data to resend

in-transit messages to be cleared. In addition, a small group of eight processes has negligible waiting time in barriers during the process. In contrary, in NORM, checkpoint time is relatively high, and is steadily increasing when scaled. It is also vulnerable to unexpected delays in checkpoints, observed as spikes in the graph. It is important to note that, even with a rather ad-hoc grouping method, as represented by GP4, the performance is considered better than NORM. With a good grouping method, group-based checkpoints could perform as efficiently as uncoordinated checkpoints. GP spent almost the same time in checkpoints with 16 to 128 processors, and is expected to have a similar behavior *when further scaled*.

Figure 6b shows the time needed in preparing a restart. Globally coordinated checkpoint is always the most efficient one, since there is no need to resend any messages. Processes may start executing shortly after the checkpoint images are loaded. Although being the fastest to checkpoint, GP1 is the slowest and the most unpredictable one in restarts, which is caused by resending variable amounts of messages to all other processes. Also, any slight delay in one process might greatly delay the whole progress in a restart. This is shown as sharp spikes in the diagram. The risk of delay becomes higher when the system is scaled. GP performs only slightly worse than NORM because only a small amount of messages have to be resent, to a small set of processes. In GP, processes do not need to consider message replays within their group, resulting in a faster operation in this stage. GP is as scalable as NORM.

Figure 7 shows the total amount of data to be resent and Figure 8 shows the actual number of resend operations to complete a restart. As shown in the Figure, GP1 is the least scalable and the figures vary more than in GP and GP4. This is because of the uncoordinated checkpoints of GP1. GP, however, is able to achieve good scalability via a good formation. GP4 scales

steadily like GP because in GP4, processes are always partitioned in 4 groups, such that the probability to resend messages would not vary a lot.

Figure 9 shows the average time spent on each stage of checkpoint, with different grouping methods, using 16 and 128 processes. With the same system scale, time spent in the actual checkpoint (*"Checkpoint"*) always remains the same as the duration is dictated by the memory usage of the application. Grouping method affects the time spent on coordination. On a small scale of 16 processes, NORM spent roughly the same time in actual checkpoint as in coordination (*"Coordination"*). When the system is scaled to 128 processes, memory usage is reduced as the problem is divided into smaller pieces, resulting in a faster *Checkpoint* stage. However the cost in coordinating the checkpoint in NORM increases so much that it becomes the dominating factor. The actual checkpoint takes only a fraction of the total. With a good grouping method, as demonstrated in GP, the overhead is kept at the minimal.

Figure 10 shows the results of a different testing scenario. Instead of doing only one checkpoint, the application is checkpointed at fixed intervals until the application finishes. The problem size N is set to 56000 to lengthen the execution time. The problem is run with 128 processes. A checkpoint interval of zero represents no checkpoint is ever made. From the figure, two important observations are made. Firstly, when no checkpoint is made GP is inevitably less efficient than NORM due to message logging. However the loss could be *compensated* when there are more checkpoints. GP caught up with NORM in terms of execution time when both GP and NORM performed four checkpoints, with a 180-second checkpoint interval. GP would outperform NORM when there are sufficient checkpoints, as shown in 60- and 120-second intervals.

Secondly, given its better performance, GP allows more checkpoints to be done, while still being more efficient than NORM in terms of total execution time. As a result the average amount of work between checkpoints can be reduced. This means if there are any restarts, the expected work loss is also reduced. As a conclusion, the proposed solution is suitable for large-scale, long-running applications.

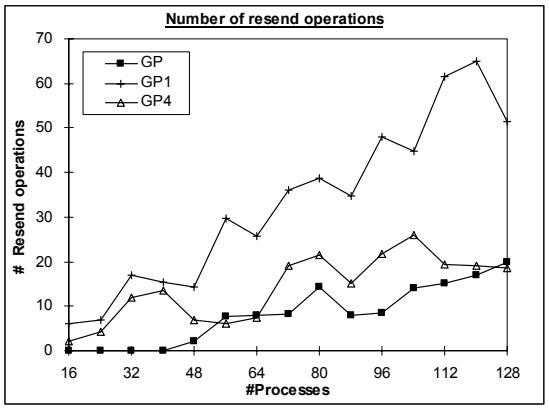


Figure 8 Number of resend operations

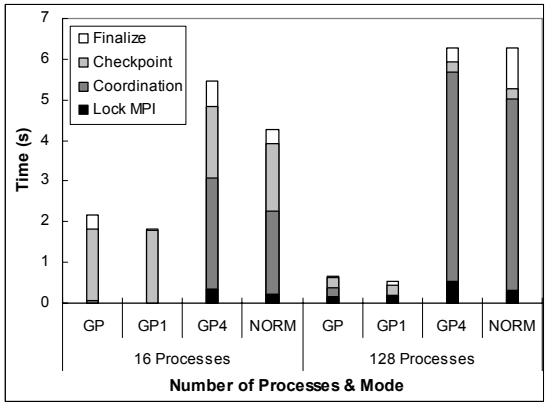


Figure 9 Checkpoint time breakdown

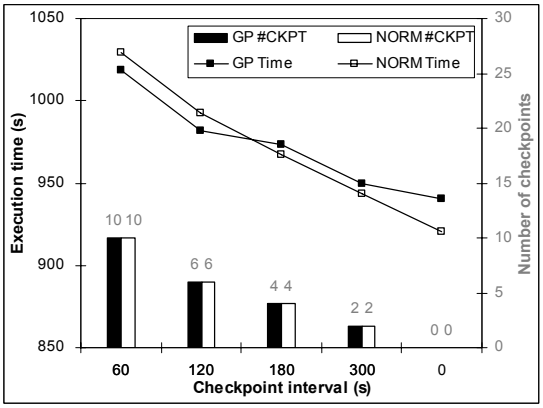
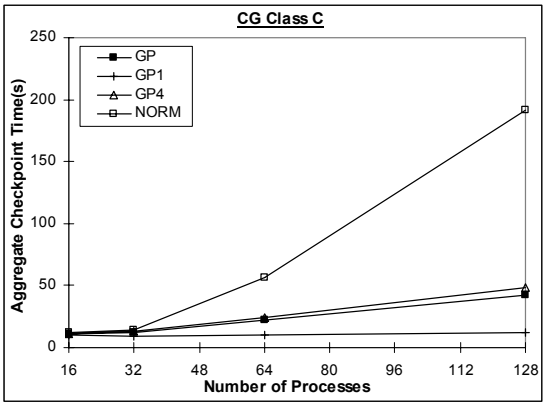
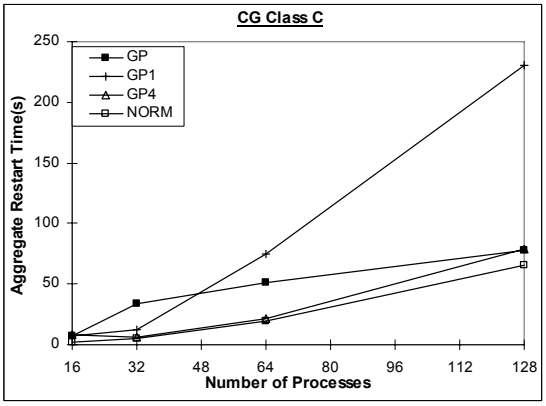


Figure 10 Effect of multiple checkpoints

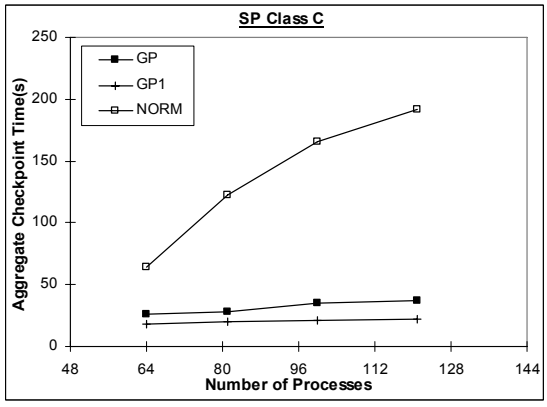


(a) Checkpoint

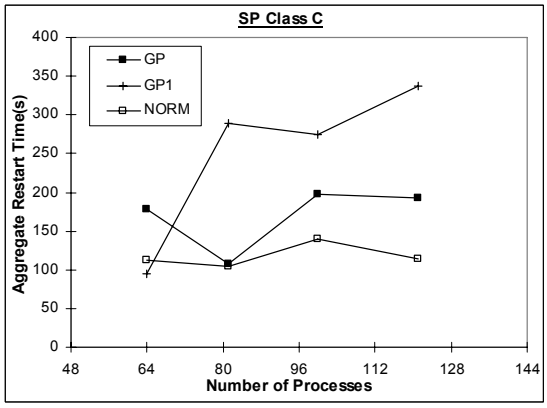


(b) Restart

Figure 11 CG: Summed checkpoint and restart time



(a) Checkpoint



(b) Restart

Figure 12 SP: Summed checkpoint and restart time

5.2. Experiments with NPB

A similar set of experiments is done using NPB, using the applications CG and SP. Problem sizes are of Class C. Due to the different restrictions on the number of processes, 16, 32, 64 and 128 processes are used for CG, for SP, 64, 81, 100 and 121 processes are used. GP4 is only tested on CG as it is not appropriate for SP's system size. MPI traces are obtained and analyzed for each case. Figure 11 and Figure 12 show the summed checkpoint and restart times for CG and SP respectively. Similar to the results with HPL, checkpoint time in GP is much better than NORM and is comparable to GP1. During restarts GP is as efficient as NORM, and less varying than GP1.

5.3. Checkpoint on remote storage and comparison with MPICH-VCL

MPICH-VCL uses non-blocking coordinated checkpointing as opposed to the blocking approach in LAM/MPI. Also, MPICH-VCL stores the checkpoint images at a remote checkpoint server rather than the local disk. In the following experiments, 4 isolated computing nodes act as the checkpoint servers for MPICH-VCL. LAM/MPI is also configured to store checkpoint images at these servers via NFS. Experiments are done using CG, Class C from 16 to 128 processes. Checkpoints are triggered every 120 seconds in MPICH-VCL, GP is then forced to take the same number of checkpoints by using a different checkpoint interval, to ensure fairness in the comparison. This is due to the difference in execution time, that GP would take fewer checkpoints with 120 second intervals. The experiment is repeated 5 times to take averaged results.

Figure 13 shows the total execution time and the number of checkpoints completed during the execution, when using remote storage for checkpoint images. GP shows a clear edge over VCL as the system scales up.

Figure 14 shows the average time required per checkpoint, again GP is more efficient than VCL throughout the experiment. Following the trend MPICH-VCL may perform much less efficiently than GP when the system is further scaled.

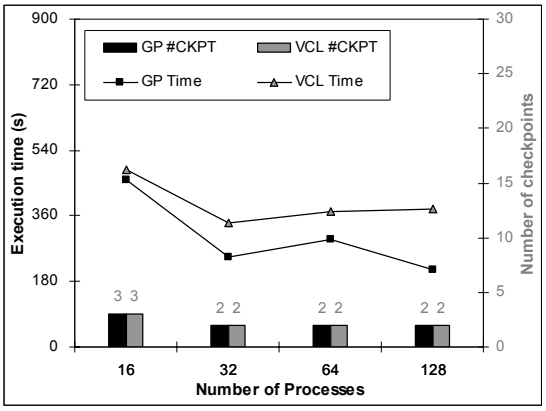


Figure 13 Effect of scale

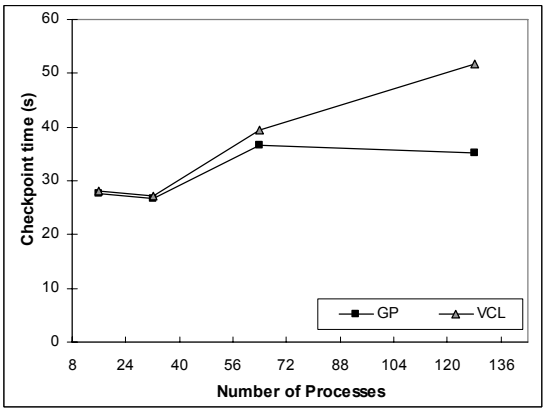


Figure 14 Average time per checkpoint

6. Related Works

LAM/MPI and MPICH are two widely used MPI implementations. LAM/MPI has blocking coordinated checkpointing support built-in. MPICH however does not support checkpointing by itself. As a result, MPICH-V projects [2] evolved to provide fault tolerant support to MPICH. As described in previous sections, LAM/MPI and MPICH-VCL suffer from scalability problems.

A recently released protocol, MPICH-PCL [4], which follows a blocking approach, is expected to have a similar behavior to LAM/MPI when applied to large-scale systems. Moreover, the developers of the MPICH-V family are aware that their protocols may add significant message overheads [2] to the original MPICH. Therefore, MPICH-V may not be efficient for large-scale systems where there would be a lot of message transfers.

There are several projects that apply the concept of process groups. Process groups may be formed according to the architecture of the system. For example, MPICH-V3 [8] is being designed for grid environments that consist of multiple clusters. Within a cluster, checkpoints can be done independently using MPICH-VCL, and consistency among the clusters is maintained by message logs. For a closely connected cluster of clusters, also known as cluster federations, the HC^{3}I project [10] works in a similar fashion. Checkpoints within one cluster is done coordinately, and among the clusters, communication induced checkpointing is used. The NCCU-MPI project [9], designed for fat-node clusters, uses coordinated checkpoints on multiple processes running within a cluster node, and logs messages transferred between the nodes. These works lack flexibility where the group formation is fixed according to the system architecture.

There are also grouping approaches based on the communication behaviors of the application being checkpointed. Gopalan and Nagarajan [7] have designed a dynamic scheme that partitions processes into smaller, mutually disjoint process groups. Individual processes or groups of processes are merged into a single group when one sends or receives a message to/from the other. Coordinated checkpoints are done within a group, complemented by message logs. With this grouping method, however, all processes may eventually form as a single group when there is a sequence of messages linking up all the processes.

The proposed solution is different from these works in that group formations can be freely controlled by the user, so that the checkpointing scheme could suit both hardware and software characteristics. Moreover, as the group formation can be freely controlled, it is possible, for example, to group processor nodes that fail more frequently, and select a shorter checkpoint interval, in order to increase tolerance to failures by reducing the amount of work loss due to restarts. The above listed works do not support such feature.

7. Conclusion and Future Work

This paper has made the following contributions. Firstly, the inadequacies of current checkpoint techniques on message passing systems are identified. The current techniques may not be sufficiently scalable and flexible to be applied in large-scale message passing systems. With the issues to be solved in mind, a novel and practical solution using group-based coordinated checkpointing and message logging is proposed and implemented. Implementation is done based on LAM/MPI, one of the widely adopted MPI implementations, which allows most MPI applications to acquire fault tolerance capabilities readily. The

proposed solution does not require global checkpoints and restarts. Through experiments, the benefit of reduced coordination time by using the group-based approach is shown. The solution is efficient during both checkpoints and restarts, and remains equally efficient when the system scales. With the flexibility in forming groups, the solution is suitable for different system architectures, such as grids, cluster federations or simple clusters. Comparing with the original LAM/MPI implementation, the proposed solution is able to perform more checkpoints within the execution, with shorter or similar total execution times. This increases the throughput of the system by reducing work loss due to rollback recovery, in addition to the reduction in coordination time. The proposed solution also performs better than MPICH-VCL by having better scalability. To assist optimal process group formation, a light weight MPI communication tracer is designed and implemented, where the trace output can be analyzed to reveal the communication behavior of the application and to suggest a group formation to be applied in checkpointing.

The performance of a group-based solution relies on the group formation. While one method is given in this paper, there may be better group forming algorithms that would need further research. For example, the change in communication pattern in different stages of the application may lead to a change in group formation. Moreover, the traces would also give a hint to select a fixed optimal checkpoint interval, or to place checkpoints at exact points of the execution.

References

[1] Lorenzo Alvisi, Sriram Rao, Syed Amir Husain, Asanka Mel de and E.N. (Mootaz) Elnozahy. *An Analysis of Communication-Induced Checkpointing*, in FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, IEEE Computer Society, pp. 242-249, 1999

[2] Aurélien Bouteiller, Thomas Hérault, Géraud Krawezik, Pierre Lemarinier and Franck Cappello. MPICH-V: a Multiprotocol Fault Tolerant MPI, International Journal of High Performance Computing and Applications, vol.20 no.3:319-333, 2006

[3] K. Mani Chandy and Leslie Lamport. *Distributed Snapshots: Determining Global States of Distributed Systems*, ACM Transactions on Computer Systems, vol.3 no.1:63-75, 1985

[4] Camille Coti, Thomas Hérault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez and Franck Cappello. *Blocking vs Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI*, in proceedings of The IEEE/ACM SC2006 Conference, 2006

[5] Jason Duell, Paul H. Hargrove and Eric Roman. *The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart*, publication LBNL-54941, Berkeley Lab Technical Report, 2002

[6] E.N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang and David B. Johnson. *A survey of rollback-recovery protocols in message-passing systems*, ACM Computing Surveys, vol.34 no.3:375-408, 2002

[7] N.P. Gopalan and K. Nagarajan. *Self-refined Fault Tolerance in HPC Using Dynamic Dependent Process Groups*, in Distributed Computing - IWDC 2005, pp. 153-158, 2005

[8] Pierre Lemarinier, Aurélien Bouteiller and Franck Cappello. *MPICH-V3: Toward a High Performance Fault Tolerant MPI for Cluster of Clusters Grid*, in Poster Section, High Performance Networking and Computing (SC2003), Phoenix, USA, 2003

[9] Wei-Jih Li and Jyh-Jong Tsay. *Checkpointing Message-Passing Interface(MPI) Parallel Programs*, in Proceedings of Pacific Rim International Symposium on Fault-Tolerant Systems, 1997, IEEE Computer Society, pp. 147-152, 1997

[10] Sébastien Monnet, Christine Morin and Ramamurthy Badrinath. *Hybrid checkpointing for parallel applications in cluster federations*, in IEEE International Symposium on Cluster Computing and the Grid, 2004, pp. 773-782, 2004

[11] Robert H.B. Netzer and Jian Xu. *Necessary and Sufficient Conditions for Consistent Global Snapshots*, IEEE Transactions on Parallel and Distributed Systems, vol.6 no.2:165-169, 1995

[12] Mamoru Ohara, Masayuki Arai, Satoshi Fukumoto and Kazuhiko Iwasaki. *Finding a Recovery Line in Uncoordinated Checkpointing*, in ICDCSW '04: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04), IEEE Computer Society, pp. 628-633, 2004

[13] A.J. Oliner, Ramendra K. Sahoo, José E. Moreira and M. Gupta. *Performance Implications of Periodic Checkpointing on Large-Scale Cluster Systems*, in IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18, IEEE Computer Society, 2005

[14] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul H. Hargrove and Eric Roman. *The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing*, International Journal of High Performance Computing Applications, vol.19 no.4:479-493, 2005

[15] Robert E. Strom and Shaula A. Yemini. *Optimistic recovery in distributed systems*, ACM Transactions on Computer Systems, vol.3 no.3:204-226, 1985

[16] High Performance Linpack: http://www.netlib.org/benchmark/hpl/

[17] NAS Parallel Benchmarks: http://www.nas.nasa.gov/Resources/Software/npb.html