Extended mpiJava for Distributed Checkpointing and Recovery

Emilio Hernández, Yudith Cardinale, and Wilmer Pereira

Universidad Simón Bolívar,
Departamento de Computación y Tecnología de la Información,
Apartado 89000, Caracas 1080-A, Venezuela
{emilio,yudith,wpereira}@ldc.usb.ve

Abstract. In this paper we describe an *mpiJava* extension that implements a parallel checkpointing/recovery service. This checkpointing/recovery facility is transparent to applications, i.e. no instrumentation is needed. We use a distributed approach for taking the checkpoints, which means that the processes take their local checkpoints independently. This approach reduces communication between processes and there is not need for a central server for checkpoint storage. We present some experiments which suggest that the benefits of this extended MPI functionality do not have a significant performance penalty as a side effect, apart from the well-known penalties related to the local checkpoint generation.

1 Introduction

Parallel checkpointing algorithms are important in parallel environments where long-term parallel processes are executed. These algorithms can be classified into two main groups:coordinated and uncoordinated, according to the approach used to coordinate the capture of local checkpoints [1], i.e. the state of single nodes. In the coordinated approach, it is necessary to synchronize all processes in order to produce a consistent (normally centralized) global checkpoint. In the uncoordinated approach, the processes take their local checkpoints independently. As a consequence, some of the checkpoints may not belong to any consistent global checkpoint. In order to reduce the number of useless checkpoints, the processes may exchange information about their checkpointing activities. This information is mainly piggy-backed on the messages sent between processes. Under this scheme, processes can take forced checkpoints to avoid orphan messages, and log in-transit messages. The algorithms based on communication-induced protocols are called quasi-synchronous [2].

In a previous work [3], we proposed a distributed checkpointing protocol as a combination of the protocols described in [4] and [5]. The protocol described in [4] logs in-transit messages in order to be able to re-send them during the recovery process. On the other hand, the protocol described in [5], focuses on avoiding orphan messages by taking checkpoints that have not been scheduled previously (forced checkpoints) based on a communication-induced checkpointing protocol. Those protocols provide fault-tolerance in asynchronous systems, and assume

that each ordered pair of processes is connected by a reliable, directed logical channel whose transmission delays are unpredictable but finite.

In this paper we describe an extension to the *mpiJava* functionality that implements the combined checkpointing protocol. This quasi-synchronous protocol was implemented on the JNI wrappers, at each point-to-point communication method (send/receive), without changing the *mpiJava* API. It means that checkpointing and recovery facilities are transparent to applications. That is, the extended *mpiJava* is used without making any changes to the application code. Additionally, we present a case study in which we implement this algorithm. We show the results of some experiments that suggest that the benefits of this extended MPI functionality does not imply a significant performance penalty.

There are several works that implement parallel checkpointing/recovery or migration schemes in combination with communication libraries such as PVM and MPI. Most of these proposals use a coordinated approach [6–9]. MPICH-V2 [10] is a fault tolerant MPICH version that implements an uncoordinated protocol. However, a centralized checkpoint server is used to control logging messages. The algorithm we implemented does not need a centralized component because each node decides independently whether a forced checkpoint is needed.

2 Extended mpiJava

We have extended *mpiJava* functionality to include the control information needed to decide when forced checkpoints have to be taken and when to log messages. We added procedures "send_protocol" and "receive_protocol" (related to the combined protocol) at each point-to-point communication method without changing their interfaces. It means that the extended *mpiJava* preserves the same *mpiJava* API. Specifically, "send_protocol" and "receive_protocol" were added at `Comm` package, where all point-to-point communication methods are defined. We implemented a new package called `Ckpt` to define and manage data structures of the combined protocol (see Figure 1). In this package there are two important procedures, `initialize` and `take_checkpoint`, which are described below.

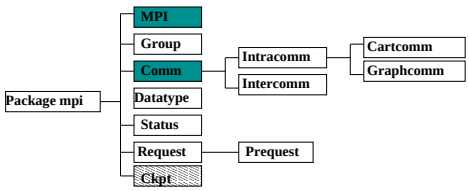

Fig. 1. Principal classes of the extended mpiJava

The `initialize` procedure is executed before any "send" or "receive". It initializes the data structures used by the checkpointing protocol. This procedure is called by MPI.Init, which is the first *mpiJava* method invoked by each **process**.

The `take_checkpoint` procedure takes the checkpoints, either normal or forced. It first increases the checkpoint number corresponding to the current process (its logical clock). Then, for each message, it checks whether the message is not an in-transit one, in which case the message is suppressed from the message log. Finally, the current state should be saved. A copy of the checkpoint number array and a copy of the log are also saved in stable storage.

The `mpiJava_send` procedure is the wrapper of `mpi_send` routine. The "send_protocol" registers the event "send" and executes the actual mpi_send routine. The current message is logged in the v_log, while procedure take_checkpoint, as explained above, identifies the messages that may become in-transit during recovery.

Procedure mpiJava_send(m, dest_rank)
$known\_received[dest\_rank][dest\_rank] := +1;$
$sent\_to[dest\_rank] := true;$
append $(m, known\_received[dest\_rank][dest\_rank], dest\_rank)$ **to** v_log
// Send message m to process $dest\_rank$
$actual\_mpi\_send(m, dest\_rank, greater, ckpt, taken, known\_received);$
end procedure

The `mpiJava_receive` procedure is the wrapper of the `mpi_receive` routine. The "receive_protocol" executes the actual mpi_receive routine and takes forced checkpoints (if necessary) to avoid orphan messages during a potential recovery. `ckpt[source_rank]` is the logical clock of process with `rank= source_rank` when message `m` was sent. This procedure is more complex because it actually checks the control information and has to consider several cases. A detailed description of this procedure follows:

Procedure mpiJava_receive (m, source_rank)
// Receives message m from process $source\_rank$
$actual\_mpi\_receive(source\_rank, m, r\_greater, r\_ckpt, r\_taken, r\_known\_received);$
// Evaluating control information to avoid orphan messages
if (∃ rank : ($sent\_to$[rank] ∧ r_greater[rank]) ∧ (ckpt[source_rank] > $ckpt[my\_rank]$) ∨
(r_ckpt[my_rank] = $ckpt[my\_rank]$ ∧ r_taken[my_rank]))
then take_checkpoint // forced checkpoint
end_if
switch
case $ckpt[source\_rank] > ckpt[my\_rank]$ **do**
$ckpt[my\_rank] := ckpt[source\_rank]; greater[my\_rank] := false;$
forall $rank \neq my\_rank$ **do** $greater[rank] := r\_greater[rank]$ **end_do**;
end case
case $ckpt[source\_rank] = ckpt[my\_rank]$ **do**
forall $rank$ **do** $greater[rank] := greater[rank] \wedge r\_greater[rank]$ **end_do**;
end case
// **case** $ckpt[source\_rank] < ckpt[my\_rank]$ **do nothing**
end_switch;
forall $rank \neq my\_rank$ **do**
switch
case $r\_ckpt[rank] > ckpt[rank]$ **then**
$ckpt[rank] := r\_ckpt[rank]; taken[rank] := r\_taken[rank];$
end case
case $r\_ckpt[rank] = ckpt[rank]$ **then**
$taken[rank] := taken[rank] \vee r\_taken[rank]$
end case

```
            // case r_ckpt[rank] < ckpt[rank] do nothing
        end_switch;
    end_do;
    JNI_deliver(m); // gives message m to the application
    known_received[my_rank, source_rank] := known_received[my_rank, source_rank] + 1;
    forall (rank_x, rank_y) do known_received[rank_x, rank_y] :=
        max(known_received[rank_x, rank_y], r_known_received[rank_x, rank_y]);
    end_do
    end procedure
```

We added a new method named `InitAndRecoverNative` at the `MPI` package, which should be called during the recovery process. `InitAndRecoverNative` is similar to `InitNative` except that the first one calls method `ReadCheckpoint`, which is in charge of re-sending all in-transit messages stored in checkpoint files. Method `ReadCheckpoint` is part of the package `Ckpt`.

3 A Case Study: Checkpointing Service on a Java-based Grid Platform

In this section we describe the checkpointing service implemented on SUMA/G[1] [11], a distributed platform that transparently executes both sequential and parallel Java programs on remote machines. It extends the Java execution model to be used on Globus-based grid platforms. The Execution Agents of SUMA/G are JVMs that can be deployed in Globus worker nodes. These agents provide checkpointing services by using an extended JVM that is able to capture local checkpoints (currently, we use the extended JVM described in [12]). There are two key components: `SUMAgCkpMonitor` and `SUMAgRecover`. The Execution Agent starts the application, as well as the `SUMAgClassloader` and the thread `SUMAgCkpMonitor` in the extended JVM. If a fault occurs, the `SUMAgRecover` is invoked for restoring the application execution from its last consistent global checkpoint.

Figure 2 shows the interactions between `SUMAgCkptMonitor` and the extended *mpiJava* wrappers in a node. This scheme is followed for all processes of the parallel application. `SUMAgCkpMonitor` periodically takes checkpoint, in an asynchronous way (step 1 in figure 2). Every time an asynchronous checkpoint is taken, `SUMAgCkptMonitor` calls method `take_checkpoint` to update and save control information (steps 2 and 3).

For each "send" or "receive" communication call executed by the application (step 4), the extended *mpiJava* wrappers update the control information (i.e, data structures used by the combined checkpointing protocol), as shown in step 5 in figure 2. In case of a "receive" call, the `receive_protocol` decides if a forced checkpoint is needed in order to avoid orphan messages (step 6). In this case, `take_checkpoint` method saves the control information and the state of application threads. To save state of application threads it is necessary to make an upcall to `SUMAgCkpMonitor` (steps 7 and 8). After `send_protocol` or `receive_protocol` is executed, the actual mpi_send or mpi_receive routine is executed (step 9).

[1] http://suma.ldc.usb.ve

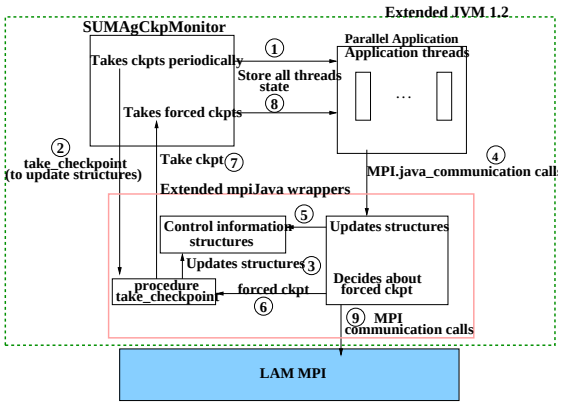

Fig. 2. Interaction between SUMA/G, Extended JVM and Extended *mpiJava* during normal execution

3.1 SUMA/G Recovery process

If a failure occurs in the platform while an application is running, an exception is caught by the `SUMAg Proxy`. SUMA/G launches the recovery algorithm described in [3] which determines the last consistent global checkpoint (step 1 in Figure 3). When a new execution platform is identified, a recovery process is initiated on each node of the parallel application. A `SUMAgRecover` thread is initiated on each node, one per process in the parallel application (step 2). Each `SUMAgRecover` reads the state of threads from checkpoint files and restores threads execution (step 3), executes method `InitAndRecoverNative` (step 4) to re-start MPI and re-send all in-transit messages (step 5). Eventually, each process will receive corresponding in-transit messages (step 6) and the execution will continue normally.

4 Experimental Results

We evaluated the overhead produced by the extended *mpiJava* wrappers on two parallel programs. We used a modest platform to make the measurements, specifically several small clusters of 143 MHz SUN Ultra 1 workstations running Solaris 7, with 64 MB Ram, connected through a 10Mbps switched Ethernet LAN. The first program, called "Pi_Number", is simple and calculates π. The processes keep the set of digits obtained during the execution, so the checkpoint size increases during the execution. The second program is a kernel of a real application, called "Acoustic_Par", which solves an acoustic wave propagation model on a homogeneous, two dimensional medium. Its checkpoint size is constant during the execution.

We measured the total execution time invoking (T_{ckp}) and without invoking (T_{nockp}) the checkpointing service. The net checkpointing overhead and checkpointing overhead percentage are given by O_{ckp} and $O\%_{ckp}$.

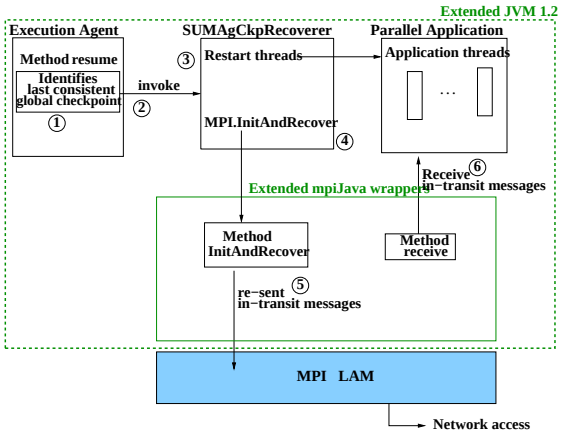

Fig. 3. Interaction between SUMA/G, Extended JVM and Extended *mpiJava* during recovery process

$$O_{ckp} = (T_{ckp} - T_{nockp}) \qquad O\%_{ckp} = \frac{(T_{ckp} - T_{nockp})}{T_{nockp}} * 100$$

In table 1 each row represents a single execution. The checkpoints were taken every 2 minutes. The overhead ($O\%_{ckp}$) exhibited when the checkpoint service is active increases with the number of processors. This is mainly due to the checkpoint calls and the distributed checkpointing protocol. Note that the T_{nockp} for the "Pi_Number" application increases as the number of processes increases. This is due to the fact that all processes roughly carry out the same amount of work, regardless of the number of processes. Even though this is not a typical parallel program, this example can help measure the checkpointing overhead. On the other hand, the execution time of "Acoustic_Par" reduces as the number of processors is increased.

In these experiments the higher checkpointing overhead is 6.11%. However, we have measured how much of that overhead is due to the extended *mpiJava*. Tables 2 and 3 show the measurements taken from the master node P_0 of each application. $T_{ThrMonitor}$ is the time taken by `SUMAgCkpMonitor` to save the state of the threads by calling extended JVM facilities. $T_{save}(F_{general})$ is the time, also taken by `SUMAgCkpMonitor`, spent on saving the checkpoint information (such as number of threads, name of main thread, thread names, etc.). The time to save the control information is $T_{actmpijava}$, while $T_{save}(\overline{M_{intransit}})$ represents the total time to log all in-transit messages in stable storage. T_{send} and $T_{receive}$ represent the accumulated time of "send_protocol" and "receive_protocol" procedures respectively during the execution. Table 2 also shows the average size of in-transit messages ($\overline{M_{intransit}}$) and total number of sent (M_{sent}) and received ($M_{received}$) messages. In case of "PI_number", P_0 does not send any message, thus we show T_{send} in $P_i \neq P_0$ only as a reference. The total overhead of extended *mpiJava* wrappers is denoted as $O_{mpiJavaExt}$ and is calculated as follows:

$$O_{mpiJavaExt} = T_{actmpijava} + T_{save}(\overline{M_{intransit}}) + T_{send} + T_{receive}$$

App name	# of proc	$T_{nockp}(1)$	$T_{ckp}(2)$	Checkpoints			O_{ckp}	$O\%_{ckp}$
				#	min size (KB)	max size (KB)	(min)	
Pi_Number	2	6.74 m	7.1 m	3	1	131	0.36	5.1%
	4	7.31 m	7.77 m	3	1	139	0.48	5.92%
	6	7.68 m	8.18 m	4	1	142	0.5	6.11%
Acoustic_Par	2	5.45 m	5.51 m	2	1	1	0.06	1.1%
	4	3.56 m	3.65 m	2	1	1	0.09	2.47%
	8	2.64 m	2.71 m	2	1	1	0.07	2.58%

Table 1. Total execution time for the parallel programs

Last column of table 3 shows $O_{mpiJavaExt}$ and its percent from total checkpointing overhead (O_{ckp}). Note that the overhead of the extended *mpiJava* only represents 15%.

App name	# of proc.	# of ckpts	$T_{ThrMonitor}$	T_{save} ($F_{general}$)	$T_{actmpijava}$	$\overline{M_{intransit}}$	$T_{save}(\overline{M_{intransit}})$
Pi_Number	2	3	274 msec	42 msec	3 msec	220B	40 msec
	4	3	392 msec	42 msec	3 msec	255B	42 msec
	6	4	405 msec	42 msec	4 msec	410B	45 msec
Acoustic_Par	2	2	43 msec	8 msec	2 msec	3KB	7 msec
	4		73 msec				
	8		54 msec				

Table 2. Overhead of Extended *mpiJava* taken from master node (P_0)

App name	# of proc.	M_{sent}	T_{send}	$M_{received}$	$T_{receive}$	O_{ckp}	$O_{mpiJavaExt}$ ($O\%_{mpiJavaExt}$)
Pi_Number	2	0	0.8 msec (on $P_i \neq P_0$)	2000	1.5 msec	360 sec	44.5 sec (12%)
	4			4000	2.9 msec	480 sec	47.9 sec (10%)
	6			6000	4.6 msec	500 sec	53.6 sec (11%)
Acoustic_Par	2	140	0.01 msec	102	0.07 msec	60 sec	9.08 sec (15%)
	4	280	0.08 msec	204	0.15 msec	90 sec	9.23 sec (10%)
	8	350	0.08 msec	274	0.09 msec	70 sec	9.17 sec (13%)

Table 3. Overhead of Extended *mpiJava* taken from master node (P_0) (cont)

5 Conclusions

We present an implementation of an uncoordinated checkpointing/recovery protocol on *mpiJava*. The implemented checkpointing/recovery facility is transparent to applications. We use a distributed approach for taking the checkpoints, which means that the processes take their local checkpoints independently. This approach reduces communication between processes and a central server for checkpoint storage is not needed. This checkpointing/recovery facility does not need any instrumentation of the source code but it needs an extended JVM for local checkpoint generation.

We have tested this extended *mpiJava* functionality and the results suggest that this approach does not involve a significant performance overhead, especially when compared with the overhead of taking the local checkpoints. However,

experiments with big parallel applications are important to corroborate these results. We expect more parallel *mpiJava* applications to become available in the public domain for continuing to test the checkpointing facility presented. In the future we are going to implement this facility on the rest of the *mpiJava* functions, such as the broadcast.

References

1. Elnozahy, E.N., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys **34**(30) (2002) 375–408
2. Manivannan, D., Singhal, M.: Quasi-Synchronous Checkpointing: Models, Characterization, and Classification. IEEE Transactions on Parallel and Distributed Systems **10**(7) (1999) 703 – 713
3. Cardinale, Y., Hernández, E.: Parallel Checkpointing Facility in a Metasystem. In: Proceedings of Parallel Computing Conference (PARCO'01), Naples, Italy (2001)
4. Mostefaoui, A., Raynal, M.: Efficient message logging for uncoordinated checkpointing protocols. Technical Report 1018, Institut de recherche en informatique et systemes aleatoires (IRISA) (1996)
5. Helary, J., Mostefaoui, A., R. Netzer, Raynal, M.: Communication-based prevention of useless checkpoints in distributed computations. Technical Report 1105, Institut de recherche en informatique et systemes aleatoires (IRISA) (1997)
6. Stellner, G.: Cocheck: Checkpointing and process migration for MPI. In: 10th International Parallel Processing Symposium. (1996)
7. Sankaran, S., Squyres, J.M., Barrett, B., Lumsdaine, A., Duell, J., Hargrove, P., Roman, E.: The Lam/Mpi Checkpoint/Restart Framework: System-Initiated Checkpointing. International Journal of High Performance Computing Applications **4**(19) (2005) 479–493
8. Zhang, Y., Xue, R., Wong, D., Zheng, W.: A Checkpointing/Recovery System for MPI Applications on Cluster of IA-64 Computers. In: ICPP 2005 Workshops. International Conference Workshops. (2005) 320–327
9. N. Woo, H. Y. Yeom, T.P.: MPICH-GF: Transparent Checkpointing and Rollback-Recovery for Grid-enabled MPI Processes. IEICE Transactions on Information and Systems, Special Section on Hardware/Software Support for High Performance Scientific and Engineering Computing **E87-D**(7) (2004) 1820–1828
10. Bouteiller, A., Cappello, F., Herault, T., Krawezik, G., Lemarinier, P., Magniette, F.: MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging. In: Proceedings of High Performance Networking and Computing (SC2003). (2003)
11. Cardinale, Y., Hernández, E.: Parallel Checkpointing on a Grid-enabled Java Platform. Lecture Notes in Computer Science **3470**(EGC2005) (2005) 741 – 750
12. Bouchenak, S.: Making Java applications mobile or persistent. In: Proceedings of 6th USENIX Conference on Object-Oriented Technologies and Systems. (2001)