9th International Conference on Parallel Processing and Applied Mathematics PPAM 2011
Torun, Poland, September 11-14, 2011

HADAB: a system enabling fault tolerance in parallel applications running in distributed environments

BOCCIA, Vania (University of Naples Federico II), CARRACCIUOLO, Luisa (CNR), LACCETTI, Giuliano (University of Naples Federico II), LAPEGNA, Marco (University of Naples Federico II), MELE, Valeria (University of Naples Federico II)

The problem

In recent decades the focus of the scientific community moved from the traditional parallel computing systems to high performance computing systems for distributed environments.

These systems, generally constituted by HPC resources (clusters) geographically scattered, provide increasing computing power and are characterized by a great resources availability (typical of distributed systems) and a high local efficiency (typical of traditional parallel systems).

These environments are characterized by high dynamicity in resources load and by a high failure rate, thus *fault tolerance* and *performance* maintainance are key issues.

For many years researchers are working to identify standard methods to solve the problem of fault tolerance and efficiency of software designed for distributed environments.

A problem solution

Design and deploy of a *checkpointing/migration* system, in order to enable *fault tolerance* in parallel applications running in a distributed environments.

Our solution

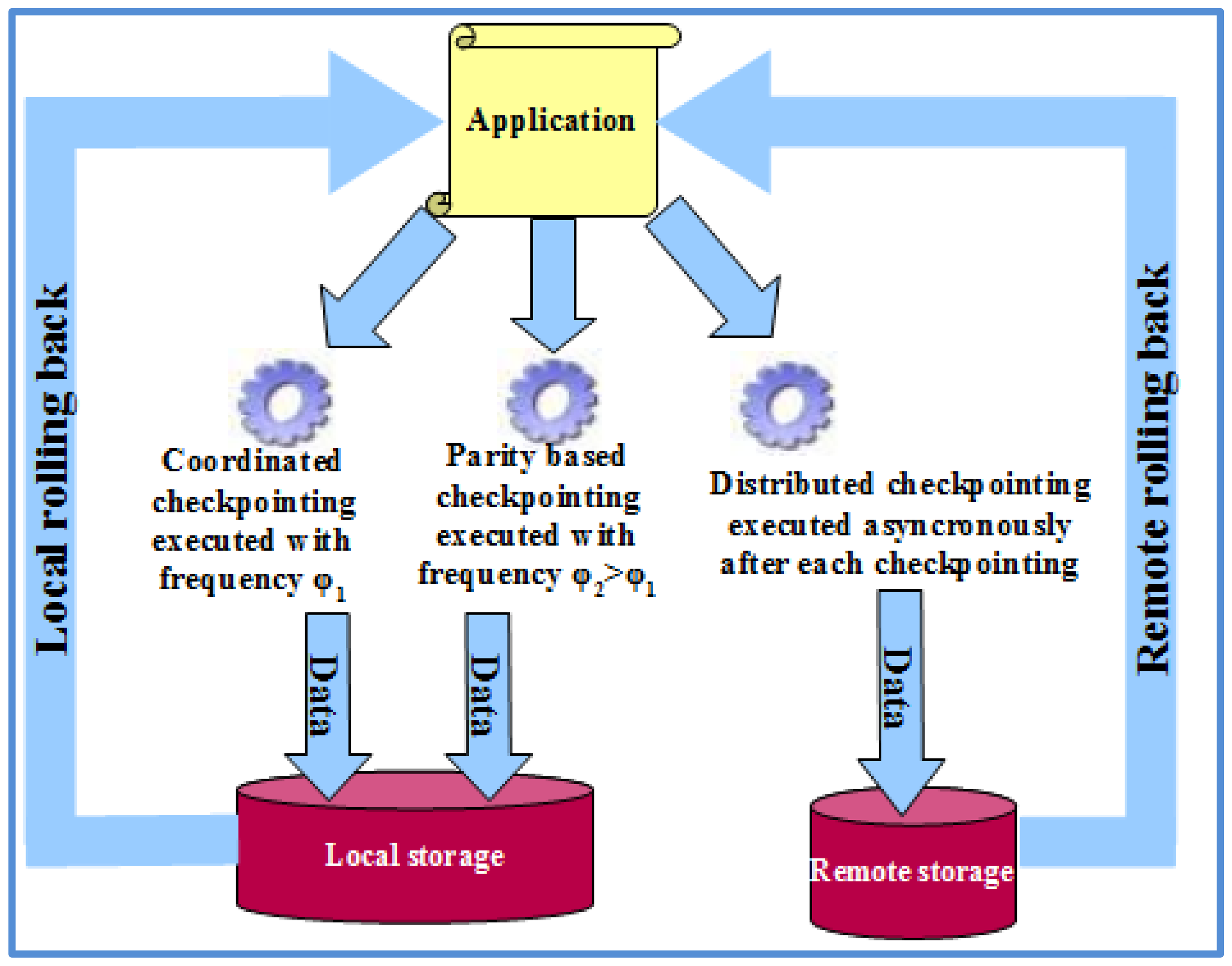
HADAB: The Hybrid, Adaptive, Distributed, Algorithm-Based checkpointing

Hybrid, because combines strategies: a disk based variant of diskless parity-based and coordinated checkpointing
Adaptive, because different checkpointing techniques are performed each with different frequency, with the aim to reduce the total overhead
Distributed, because checkpointing data are periodically saved on a remote storage resource
Algorithm-based because, although hard to implement, this approach is still the safest method to select and reduce the checkpointing data amount.

HADAB checkpointing strategy

1. Parity based strategy introduces a lower I/O overhead than a non-coding one, but it can tolerate only one fault at a time.
2. Coordinated checkpointing, instead, can tolerate up to *p-1* (where *p* is the number of processes) faults at a time, but it is more expansive in terms of I/O overhead.
3. For this reason in our strategy each checkpointing is executed with a rate that depends on the extimated execution time in a way to not excessively increase the total checkpointing overhead.

Combining these two strategies it is possible to survive up to *p-1* faults but not to the checkpointing processor fault. To avoid this point of failure, after a coordinated checkpointing, we save local checkpointing data on a remote storage in asyncrosous way. Thus the application can survive also to *p* faults and the total I/O overhead due to checkpointing doesn't encrease.



The application starts with φ_1 and φ_2 default values. During the execution those values are changed on the bases of estimated execution time in a way to not encrease the total I/O overhead.

A case study: HADAB deployment on the PETSc Conjugate Gradient

We deployed HADAB strategy into the parallel version of conjugate gradient (CG) algorithm implemented in PETSc library with the following two objectives:

- to achieve a fault tolerant version of the PETSc CG, through the use of HADAB checkpointing;
- to realize, in a distributed environment, a system to enable migration of CG-based applications on alternative resources, when faults occur.

In order to develop a fault tolerant version of CG algorithm, we follow an algorithm-based approach: we add to the PETSc CG routine, the code needed to implement checkpointing and rolling back phases of the HADAB strategy.

Fault tolerant version of Conjugate Gradient with hybrid adaptive distributed checkpointing: code fragment

```
PetscErrorCode KSPSolve_CGFT(KSP ksp)
PetscFunctionBegin;
/* initialization phase */
...
if (restart) then
   rt = CheckCheckpoint(...);
   if (rt == 1) then
      ierr = PetscRollbackCoord(...);
   else if (rt == 0) then
      ierr = PetscRollbackCodif(...);
   else
      printf(It is impossible to recover from the fault);
   end if
end if
repeat
   ...
   /*  main iteration cycle of the CG algorithm  */
   ...
   if (chkenable) then
      /* ck_coord is the iteraction number when
      coordinated checkpointing is performed */
      /* ck_codif is the iteraction number when
      coded checkpointing is performed */
      if (i % ck_coord == 0) then
         ierr = PetscCheckpointingCoord(...);
         ierr = PetscStartCopyThreads(...);
      else/* caso i % ck_codif == 0 */
         ierr = PetscCheckpointingCodif(...);
         ierr = PetscStartCollectThreads(...);
      end if
      ierr = PetscCheckFreq(...);
   end if
until (i < max_it && r > tol)
/* finalization phase */
PetscFunctionReturn(0);
```

About the recovery phase, the ***CheckCheckpoint*** routine determines, between the two types (coordinated or parity-based), the checkpointing that allows to restore the application from the highest iteration.

About the checkpointing phase, with a frequency respectively equal to *ck_codif* for parity-based and *ck_coord* for coordinated checkpointing, **PetscCheckpointingCodif** and **PetscCheckpointingCoord** routines are executed.

PetscStartCopyThreads and **PetscStartCollectThreads** perform the asynchronous distributed checkpointing data saving on storage resources external to the execution cluster, usefull if application have to migrate on an alternative resource.

The **PetscCheckFreq** routine modifies the checkpointing frequency on the bases of both: the average of past iterations execution time and the real data saving duration for each checkpoiting type

Some results and conclusions

- Tests are related with the solution, by 6892 iterations of CG algorithm, of a linear system where the size N of the sparse matrix is $3.9 \cdot 10^{17}$.
- All tests are performed on the HPC computational resources available at the University of Naples Federico II by S.Co.P.E. GRID infrastructure.
- Data are stored on a global scratch area based on Lustre parallel file system.

It_{fault}	T_{it} lost (secs)	T^{noC}_{tot} (secs)
1000	5460	37630+5460 = 43090
2000	10920	37630+10920 = 48550
3000	16380	37630+16380 = 54010
4000	21840	37630+21840 = 59470
5000	27300	37630+27300 = 64930
6000	32760	37630+32760 = 70390

Table 2 - Application without checkpointing mechanisms: T_{it} lost is the time spent to execute again the iterations before the fault when it occurs at iterations: 1000, 2000, 3000, 4000, 5000, 6000. T^{noC}_{tot} is the time sum of failure free total time of application execution and Tit lost.

N	Execution Time (secs)	Checkpointing Time (secs)	Total Time (secs)	Overhead % checkp.
$3.9 \cdot 10^{17}$	37630	18666	56296	49.6%

Table 1 - Application execution with checkpointing mechanisms enabled: overhead introduced in a failure free execution

It_{fault}	T_{it} lost (secs)	T^{C}_{tot} (secs)	$Overhead_{chkp}$
1000	(6 it.) 32.76	56296+32.76 = 56328.76	31%
2000	(12 it.) 65.52	56296+65.52 = 56361.52	16%
3000	(4 it.) 21.84	56296+21.84 = 56317.84	0%
4000	(10 it.) 54.60	56296+54.60 = 56350.60	-1%
5000	(2 it.) 10.92	56296+10.92 = 56306.92	-13%
6000	(9 it.) 43.68	56296+43.68 = 56339.68	-20%

Table 3 - Application with checkpointing mechanisms: T_{it} lost is the time spent to execute again only iterations before the fault and from the last saved checkpointing. In last column we report

$$Overhead_{chkp} = (T^{C}_{tot} - T^{noC}_{tot}) / T^{noC}_{tot}$$

From Table 1 we can observe that checkpointing mechanisms add about the 50% of overhead on the total execution time in absence of faults. However, if we consider execution with faults, the presence of checkpointing mechanisms becomes ever more affordable when the iteration number where the fault occurs increases (see last column of Table 3).

Conclusions

The integration of the mechanisms for fault tolerance, in libraries of scientific software as PETSc, is a *"good investment"* because fault tolerance property of the library modules are inherited by all applications that use them.

The work gave us the chance to test the quality of hybrid strategies in the implementation of mechanisms for checkpointing/migration, even by using disk-based approaches

About the overhead introduced by the check-pointing mechanisms, it is related to an application that, on a big amount of data, performs a small amount of computations. Thus checkpointing mechanisms advisability is much more evident for applications handling the same amount of data but using algorithms with more complexity than that here considered.

References:

- H.D. Simon, M.A. Heroux, P. Raghavan. *Faul Tolerance in Large Scale Scientific Computing*, Chapter 11, pages 203-220. Siam Press, 2006.
- Edward Hung and M. Phil Student. *Fault Tolerance and Checkpointing Schemes for Clusters of Workstations*. 2008.
- Lu Moura Silva and Gabriel Joao Silva. *The Performance of Coordinated and Independent Checkpointing*. Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing, pages 280-284, Washington, DC, USA, 1999. IEEE Computer Society.
- Satish Balay and William D. Gropp and Lois Curfman McInnes and Barry F. Smith. *Efficient Management of Parallelism in Object Oriented Numerical Software Libraries*. Book proceedings "Modern Software Tools in Scientific Computing", 1997.