The parallel applications for distributed systems

Introduction

Distributed systems and computational Grids (Foster and Kesselman) involve large system dynamics that it is highly desirable to reconfigure executing applications in response to the change in environments. Since parallel applications execute on large number of shared systems, the performance of the applications will be degraded if there is increase in external load on the resources caused by other applications.

Also, it is difficult for users of parallel applications to determine the amount of parallelism for their applications and hence may want to determine the amount of parallelism by means of trial-and-error experiments. Due to the large number of machines involved in the distributed computing systems, the mean single processor failure rate and hence the failure rate of the set of machines where parallel applications are executing are fairly high (Beguelin et al.) Hence, for long running applications involving large number of machines, the probability of successful completions of the applications is low. Also, machines may be removed from executing environment for maintenance.

In the above situations, it will be helpful for the users or the scheduling system to stop the executing parallel application and continue it possibly with a new configuration in terms of the number of processors used for the execution. In cases of the failure of the application due to non-deterministic events, restarting the application on a possibly new configuration also provides a way of fault tolerance. Paper defines the following terms that are

commonly used in the literature to describe parallel applications with different capabilities.

1. Moldable applications – Parallel applications that can be stopped at any point of execution but can be restarted only on the same number of processors.

2. Malleable applications – Parallel applications that can be stopped at any point of execution and can be restarted on a different number of processors. These applications are also called reconfigurable applications.

3. Migratable applications – Parallel applications that can be stopped at any point of execution and can be restarted on processors in a different site, cluster or domain.

Reconfigurable or malleable and migratable applications provide added functionality and flexibility to the scheduling and resource management systems for distributed computing.

In order to achieve starting and stopping of the parallel applications, the state of the applications have to be checkpointed. Some scholars (Elonazhy; Plank) have surveyed several checkpointing strategies for sequential and parallel applications. Checkpointing systems for sequential (Tannenbaum and Litzkow). and parallel applications (Dikken et al.) have been built. Checkpointing systems are of different types depending on the transparency to the user and the portability of the checkpoints. Transparent and semi-transparent checkpointing systems (Tannenbaum and Litzkow) hide the details of checkpointing and restoration of saved states from the users, but

are not portable. Non-transparent checkpointing systems (Geist et al.) involves the users to make some modifications to their programs but are highly portable across systems. Checkpointing can also be implemented at the kernel level or user-level.

This paper describes a checkpointing infrastructure that helps in the development and execution of malleable and migratable parallel applications for distributed systems. The infrastructure consists of a user-level semi-transparent checkpointing library called SRS (Stop Restart Software) and a Runtime Support System (RSS). The SRS library is semi-transparent because the user of the parallel applications has to insert calls in his program to specify the data for checkpointing and to restore the application state in the event of a restart.

But the actual storing of checkpoints and the redistribution of data in the event of a reconfiguration are handled internally by the library. Though there are few checkpointing systems that allow changing the parallelism of the parallel applications (Geist et al.), the system is unique in that it allows for the applications to be migrated to distributed locations with different file systems without requiring the users to manually migrate the checkpoint data to distributed locations. This is achieved by the use of a distributed storage infrastructure called IBP (Plank et al.) that allows the applications to remotely access checkpoint data.

SRS Checkpointing Library

SRS (Stop Restart Software) is a user-level checkpointing library that helps to make iterative parallel MPI message passing applications reconfigurable.

Iterative parallel applications cover a broad range of important applications including linear solvers, heat-wave equation solvers, partial differential equation (PDE) applications etc. The SRS library has been implemented in both ? and Fortran and hence SRS functions can be called from both ? and Fortran MPI programs. The SRS library consists of 6 main functions:

1. SRSJnit,

2. SRSJtestart-Value,

3. SRS-Read,

4. SRS-Register,

5. SRS-CheckJ3top and

6. SRS-Finish.

The user calls SRS-Init after calling MPIJnit. SRS-Init is a collective operation and initializes the various data structures used internally by the library. SRSJnit also reads various parameters from a user-supplied configuration file. These parameters include the location of the Runtime Support System (RSS) and a flag indicating if the application needs periodic checkpointing. SRSJnit, after reading these parameters, contacts the RSS and sends the current number of processes that the application is using. It also receives the previous configuration of the application from the RSS if the application has been restarted from a previous checkpoint.

In order to stop and continue an executing application, apart from checkpointing the data used by the application, the execution context of the

application also needs to be stored. For example, when the application is initially started on the system, various data needs to be initialized, whereas when the application is restarted and continued, data needs to be read from a checkpoint and the initialization phase can be skipped. Most checkpointing systems [30] restore execution context by storing and retrieving execution stack. This solution compromises on the portability of the checkpointing system.

Since the main goal of the SRS library is to provide heterogeneous support, the task of restoring the execution context is implemented by the user by calling SRS-Restart_Value. SRS-Restart-Value returns 0 if the application is starting its execution and 1 if the application is continuing from its previous checkpoint. By using these values returned by SRS_Restart_Value, the user can implement conditional statements in his application to perform certain parts of the code when the application begins certain other parts of the code and its execution when the application is continued from its preceding checkpoint.

SRS library uses Internet Backplane Protocol (IBP) (Plank et al.) for storage of the checkpoint data. It depots are started on all the machines the user wants to use for the application’ execution. SRS-Register is used to spot the data that will be checkpointed by the SRS library through periodic checkpointing or when SRS-Check_Stop is called. Only the data that are passed in the SRS-Register call are checkpointed. The user specifies the parameters of the data including the size, data type and data distribution when calling SRS-Register. The data distributions supported by the SRS library include common data distributions like block, cyclic and block-cyclic distributions. For

checkpointing data local to a process of the application or for data without distribution, a distribution value of 0 can be specified. SRS-Register stores the various parameters of the data in a local data structure. SRS-Register does not perform actual checkpointing of the data.

SRS_Read is the main function that achieves reconfiguration of the application. When the application is stopped and continued, the checkpointed data can be retrieved by invoking SRS-Read. The user specifies the name of the checkpointed data, the memory into which the checkpointed data is read and the new data distribution when calling SRS-Read. The data distribution specified can be conventional distributions or 0 for no distribution or SAME if the same data has to be propagated over all processes. The value SAME is useful for retrieving iterator values when all the processes need to start execution from the same iteration. The SRS-Read contacts the RSS and retrieves the previous data distribution and the location of the actual data. If no distribution is specified for SRS-Read, each process retrieves the entire portion of the data from the corresponding IBP depot used in the previous execution.

If SAME is used for the data distribution, the first process reads the data from the IBP depot corresponding to the first process in the previous execution and broadcasts the data to the other processes. If data distribution is specified in SRS_Read, SRS-Read determines the data maps for the old and new distributions of the data corresponding to the previous and the current distributions. Based on the information contained in the data maps, each process retrieves its portion of data from the IBP depots containing the data portions. Thus reconfiguration of the application is achieved by using

different level of parallelism for the current execution and specifying a data distribution in SRS-Read that may be different from the distribution used in the previous execution.

SRS-Check_Stop is a collective operation and called at various phases of the program to check if the application has to be stopped. If SRS-Check-Stop returns 1, then an external component has requested for the application to stop, and the application can execute application-specific code to stop the executing application. SRS-Check_Stop contacts the RSS to retrieve a value that specifies if the application has to be stopped.

If an external component has requested for the application to be stopped, SRS-Check-Stop stores the various data distributions and the actual data registered by SRS-Register to the IBP (Plank et al.) depots. Each process of the parallel application stores its piece of data to the local IBP depot. By storing only the data specified by SRS-Register and requiring each process of the parallel application to the IBP depot on the corresponding machine, the overhead incurred for checkpointing is significantly low. SRS-Check-Stop sends the pointers for the checkpointed data to RSS and deletes all the local data structures maintained by the library.

SRS-Finish is called collectively by all the processes of the parallel application before MPI-Finish in the application. SRS-Finish deletes all the local data structures maintained by the library and contacts the RSS requesting the RSS to terminate execution. Apart from the 6 main functions, SRS also provides SRS-DistributeFunc-Create and SRS_DistributeMap-Create

to allow the user specify his own data distributions instead of using the conventional data distributions provided by the SRS library.

Runtime Support System (RSS)

RSS is a sequential application that can be executed on any machine with which the machines used for the execution of actual parallel application will be able to communicate. RSS exists for the entire duration of the application and spans across multiple migrations of the application. Before the actual parallel application is started, the RSS is launched by the user. The RSS prints out a port number on which it listens for requests. The user fills a configuration file called srs. config with the name of the machine where RSS is executing and the port number printed by RSS and makes the configuration file available to the first process of the parallel application. When the parallel application is started, the first process retrieves the location of RSS from the configuration file and registers with the RSS during SRS_Init. The RSS maintains the application configuration of the present as well as the previous executions of the application.

The RSS also maintains an internal flag, called stop-flag that indicates if the application has to be stopped. Initially, the flag is cleared by the RSS. A utility called stop-application is provided and allows the user to stop the application. When the utility is executed with the location of RSS specified as input parameter, the utility contacts the RSS and makes the RSS set the stop-flag. When the application calls SRS-Check. Stop, the SRS library contacts the RSS and retrieves the stop-flag. The application either continues executing or stops its execution depending on the value of the flag.

When the SRS-Check_Stop checkpoints the data used in the application to IBP depots, it sends the location of the checkpoints and the data distributions to the RSS. When the application is later restarted, it contacts the RSS and retrieves the location of the checkpoints from the RSS. When the application finally calls SRS-Finish, the RSS is requested by the application to terminate itself. The RSS cleans the data stored in the IBP depots, deletes its internal data structures and terminates.

Related Work

Checkpointing parallel applications have been extensively studied (Elnozahy et al.) and have been developed checkpointing systems for parallel applications (Dikken et al., Godard et al.). Some of the systems were developed for homogeneous systems (Russ et al.) while some checkpointing systems permits applications to be checkpointed and restarted on heterogeneous systems (Naik et al.) Calypso and Plinda (Jeong et al.; Baratloo et al.) require application writers to write their programs in terms of special constructs and cannot be used with third-party software.

Systems including Dynamic PVM (Dikken et al.) and CUMULVS (Geist et al.) use PVM mechanisms for fault detection and process spawning and can only be used with PVM environments. Cocheck and Starfish (Stellner; Agbaria and Friedman) provide fault tolerance with their own MPI implementations and hence are not suitable for distributed computing and Grid systems where the more secure MPICH-G (Foster and Karonis) is used. CUMULVS (Geist et al.) Dome (Arabe et al.), the work by Hofmeister and Deconick, DRMS (Naik et al.) and DyRecT (Godard et al.) are closely related to our research in terms of

the checkpointing API, the migrating infrastructure and reconfiguration capabilities.

The CUMULVS (Geist et al.) API is very similar to our API in that it requires the application writers to specify the data distributions of the data used in the applications and it provides support for some of the commonly used data distributions like block, cyclic etc. CUMULVS also supports stopping and restarting of applications. But the applications can be stopped and continued only on the same number of processors. Though CUMULVS supports MPI applications, it uses PVM as the base infrastructure and hence poses the restriction of executing applications on PVM.

Dome (Arabe et al.) supports reconfiguration of executing application in terms of changing the parallelism for the application. But the data that can be redistributed for reconfiguration have to be declared as Dome objects. Hence it is difficult to use Dome with third-party software like ScaLAPACK where native data is used for computations. Also Dome uses PVM as the underlying architecture and cannot be used for message passing applications.

The work by Hofmeister supports reconfiguration in terms of dynamically replacing a software module in the application, moving a module to a different processor and adding or removing a module to and from the applications. But the package by Hofmeister only works on homogeneous systems. The work by Deconinck is similar to SRS in terms of the checkpointing API and the checkpointing infrastructure. Their checkpoint control layer is similar to our RSS in terms of managing the distributed data

and the protocols for communication between the applications and the checkpoint control layer is similar to ours. By using architecture-independent checkpoints, the checkpoints used in their work are heterogeneous and portable. But the work by Deconick does not support reconfiguration of application in terms of varying the parallelism for the applications.

The DyRecT (Godard et al.) framework for reconfiguration allows dynamic reconfiguration of applications in terms of varying the parallelism by adding or removing the processors during the execution of parallel application. The user-level checkpointing library in DyRecT also supports the specification of data distribution. The checkpoints are system-independent and MPI applications can use the checkpointing library for dynamic reconfiguration across heterogeneous systems. But DyRecT uses LAM MPI for implementing the checkpointing infrastructure to use the dynamic process spawning and fault detection mechanisms provided by LAM. Hence DyRecT is mainly suitable for workstation clusters and not distributed and Grid systems where the more secure MPICH-G is used (Foster and Karonis). Also, DyRecT requires the machines to share a common file system and hence applications cannot be migrated and reconfigured to distributed locations that do not share common file systems.

The DRMS (Naik et al.) checkpointing infrastructure uses DRMS programming model to support checkpointing and restarting parallel applications on different number of processors. It uses powerful checkpointing mechanisms for storing and retrieving checkpoint data to and from permanent storage. It is the closest related work to SRS in that it supports a flexible checkpointing API for reconfiguring MPI message passing applications implemented on any

MPI implementations to be reconfigured on heterogeneous systems. But DRMS also does not support migrating and restarting applications on environments that do not share common file systems with the environments where the applications initially executed. A more recent work by Kale et. al achieves reconfiguration of MPI-based message passing programs. But reconfiguration is achieved by using a MPI implementation called AMPI that is less suitable to Grid systems than MPICH-G.

Conclusions and Future Directions

In this paper, a checkpointing infrastructure for developing and executing malleable and migratable parallel applications across heterogeneous sites was explained. The SRS API has limited number of functions for seamlessly enabling parallel applications malleable. The uniqueness of the SRS system is achieved by the use of IBP distributed storage infrastructure. Discussion was shown to evaluate the overhead incurred by the applications and the times for storing, reading and redistributing checkpoints. The discussion shows that SRS can enable reconfigurability of the parallel applications with limited overhead.

One of the future main goals may be to use precompiler technologies to restore the execution context and to relieve the user from having to make major modifications in his program to provide malleability of his applications. Other future investigations can include support for checkpointing files, complex pointers and structures and to provide support for different kinds of applications.

Although the design of the checkpointing framework supports migration of heterogeneous environments, the current implementation stores the checkpoint data as raw bytes. This approach can lead to misinterpretation of the data by the application if, for example, the data is stored on a Solaris system and read by a Linux machine. This is due to the different byte orderings and floating point representations followed on different systems.

It may be useful to extend the RSS daemon to make it fault-tolerant by periodically checkpointing its state so that the RSS service can be migrated across sites. Presently, all the processes of the parallel application communicate with a single RSS deamon. This may pose a problem for the scalability of the checkpointing system, especially when large number of machines are involved. The future plan may be to implement a distributed RSS system to provide scalability.

References

LAM-MPI. http://www. lam-mpi. org.

I. Foster and C. Kesselman eds. The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, ISBN 1-55860-475-8, 1999.

M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message Passing Systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.

L. Dikken, F. van der Linden, J. J. J. Vesseur, and P. M. A. Sloot. DynamicPVM: Dynamic Load Balancing on Parallel Systems. In Wolfgang Gentzsch and Uwe

Harms, editors, Lecture notes in computer science 797, High Performance Computing and Networking, volume Proceedings Volume II, Networking and Tools, pages 273-277, Munich, Germany, April 1994. Springer Verlag.

James S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, 1997.

T. Tannenbaum and M. Litzkow. The condor distributed processing system. Dr. Dobb’s Journal, pages 40-48, February 1995.

G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications. International Journal of High Performance Computing Applications, 11 (3): 224-236, August 1997.

M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message Passing Systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.

S. H. Russ, B. K. Flachs, J. Robinson, and B. Heckel. Hector: Automated Task Allocation for MPI. In Proceedings of IPPS ’96, The 10th International Parallel Processing Symposium, pages 344-348, Honolulu, Hawaii, April 1996.

V. K. Naik, S. P. Midkiff, and J. E. Moreira. A checkpointing strategy for scalable recovery on distributed parallel systems. In Super Computing (SC) ’97, San Jose, November 1997.

A. Jeong and D. Shasha. P. Linda 2. 0: A Transactional/Checkpointing Approach to Fault Tolerant Linda. In Proceedings of the 13th Symposium on Reliable Distributed Systems, pages 96-105. IEEE, 1994.

A. Baratloo, P. Dasgupta, and Z. M. Kedem. CALYPSO: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms. In Proc. of the Fourth IEEE Intel Symp. on High Performance Distributed Computing (HPDC-4), pages 122-129, August 1995.

L. V. Kale, S. Kumar, and J. DeSouza. A Malleable-Job System for Timeshared Parallel Machines. In 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002), May 2002.

E. Godard, S. Setia, and E. White. DyRecT: Software Support for Adaptive Parallelism on NOWs. In in IPDPS Workshop on Runtime Systems for Parallel Programming, Cancun, Mexico, May 2000.

I. Foster and N. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. In Proceedings of SuperComputing 98 (SC98), 1998

J. N. C. Arabe, A. B. B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: Parallel Programming in a Heterogeneous Multi-User Environment. Supercomput-ing, 1995.

A. Agbaria and R. Friedman. Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. In In the 8th IEEE International Symposium on High Performance Distributed Computing, pages 167-176, August 1999.

G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In Proceedings of the 10th International Parallel Processing Symposium (IPPS ’96), pages 526-531, Honolulu, Hawaii, 1996.

G. Deconinck and R. Lauwereins. User-Triggered Checkpointing: System-Independent and Scalable Application Recovery. In Proceedings of 2nd IEEE Symposium on Computers and Communications (ISCC97), pages 418-423, Alexandria, Egypt, July 1997.

C. Hofmeister and J. M. Purtilo. Dynamic Reconfiguration in Distributed Systems : Adapting Software Modules for Replacement. In Proceedings of the 13th International Conference on Distributed Computing Systems, Pittsburgh, USA, May 1993.