Available online at www.sciencedirect.com
ScienceDirect
Journal of Computer and System Sciences 73 (2007) 1240–1251
JOURNAL OF COMPUTER AND SYSTEM SCIENCES
www.elsevier.com/locate/jcss

Towards highly available and scalable high performance clusters

Azzedine Boukerche [a,*], Raed A. Al-Shaikh [a,b], Mirela Sechi Moretti Annoni Notare [c]

[a] *Paradise Research Laboratory, Site, University of Ottawa, Canada*
[b] *EXPEC Computer Center (ECC), Saudi Aramco, Saudi Arabia*
[c] *Barddal University, Brazil*

Received 3 October 2005; received in revised form 11 March 2006

Available online 24 February 2007

Abstract

In recent years, we have witnessed a growing interest in high performance computing (HPC) using a cluster of workstations. This growth made it affordable to individuals to have exclusive access to their own supercomputers. However, one of the challenges in a clustered environment is to keep system failure to the minimum and to achieve the highest possible level of system availability. High-Availability (HA) computing attempts to avoid the problems of unexpected failures through active redundancy and preemptive measures. Since the price of hardware components are significantly dropping, we propose to combine both HPC and HA concepts and layout the design of a HA–HPC cluster, considering all possible measures. In particular, we explore the *hardware* and the *management* layers of the HA–HPC cluster design, as well as a more focused study on the parallel-*applications* layer (i.e. FT-MPI implementations). Our findings show that combining HPC and HA architectures is feasible, in order to achieve HA cluster that is used for High Performance Computing.

Keywords: HPC; High performance computing; High performance clusters

1. Introduction

The growth of cluster computing in recent years has primarily been encouraged by the price/performance measure. This advancement made it affordable to individuals to have exclusive access to their own supercomputers. We define computer clusters are scalable interconnected machines that are based on commodity hardware. The hardware can be any of a number of mass-market, stand-alone workstations as simple as two networked computers or as complex as thousands of nodes connected by a high-speed, low-latency network. These clusters are usually equipped with open source operating systems, such as Linux or Sun Solaris OS [1].

Commodity-hardware clusters offer several advantages over the traditional supercomputers. First, High Performance Clusters are intended to be a cheaper replacement for the more complex/expensive supercomputers to run traditional technical applications such as simulations, biotechnology, financial market modeling, data mining and stream processing [1]. Second, cluster computing can scale to very large systems. Hundreds or even thousands of machines can be networked to suit the application needs. In fact, the entire Internet can be viewed as one truly huge cluster [2].

* Corresponding author.
E-mail addresses: boukerch@site.uottawa.ca (A. Boukerche), raed.shaikh@aramco.com (R.A. Al-Shaikh), mirela@barddal.br (M.S.M.A. Notare).

0022-0000/$ – see front matter
doi:10.1016/j.jcss.2007.02.011

The third advantage is availability; replacing a "faulty node" within a cluster is trivial compared to fixing a faulty SMP component, resulting in a lower mean-time-to-repair (MTTR) for carefully designed cluster configuration [4].

On the other hand, HPCs are not problems-free. First, clusters have higher network latency with a lower network bandwidth compared to SMP and supercomputers. However, as we will see in Section 3, this network difference is becoming insignificant, thanks to the advancements in the network interconnects. The second potential problem is the frequency of hardware failures (Mean-time-to-failure, or MTTF). Because of many heterogeneous commodity hardware involved to build an HPC cluster, the probability of a hardware failure is higher than an SMP machine. Therefore, most clusters are unable to handle runtime system configuration changes caused by transient failures and require a complete restart of the entire machine [3].

2. Motivation

One of the challenges in an HPC clustered environment is to keep system failure to the minimum and to provide the highest possible level of system availability. Due to the fact that very large and complex applications are being run on increasingly larger scale distributed computing environments, High-Availability (HA) computing has become critically important to the fundamental concept of High Performance Computing (HPC). This is because commodity hardware is employed to construct these clusters, and to a certain extent, the application code's runtime exceeds the hardware's mean-time-between-failures (MTBF) rate for the entire cluster [2]. Thus, in order to efficiently run these very large and complex applications, HA computing techniques must be employed in the HPC environment.

In this paper, we combine both HPC and HA models and layout the design of a HA–HPC cluster, considering all possible measures. In particular, we explore the *hardware* and the *management* layers of the HA–HPC cluster design, as well as a more focused study on the parallel-*applications* layer (i.e. FT-MPI implementation [9–11,19]).

The rest of the paper is organized as follows. In the next section, we describe the general architecture of the Beowulf cluster, which is becoming the standard design of any HPC cluster. In Section 4, we demonstrate the modifications needed in the hardware level, in order to make a large-scale Beowulf architecture fault tolerant. In Section 5, we discuss the fault tolerant issues in the management level by demonstrating the High Availability Open Source Cluster Application Resource (HA-OSCAR) that aims toward non-stop services in the HPC environment. Next and in Section 6, we explore and compare, in more details, the fault tolerant techniques in the application level, mainly the MPI layer. In Section 7, we report our performance and test results. The last section states our conclusions and summary.

3. The Beowulf cluster architecture

Originally developed at NASA, Beowulf clusters are developed worldwide to support scientific computing [1]. Fundamentally, it is a design for HPC clusters on inexpensive personal computer hardware. A typical "large-scale" Beowulf cluster consists of the following major system components (Fig. 1):

(1) *A master (or control) server*: a master server is responsible for serving user requests and distributing them to clients via scheduling/queuing software, such as Portable Batch System (PBS) [1] or LSF [4]. To control access to the cluster, users are only permitted to access this node and are blocked from accessing the other nodes. In a simple Beowulf cluster, the master node is considered to be a single point of failure. In particular, a master node failure can render the entire cluster unusable. Therefore, redundancy should exist in order to provide HA to the master node, as we will see in the next section.
(2) *Multiple identical client (compute) nodes*: these clients or compute nodes are normally dedicated to computation. Normally, users are blocked from direct access to these compute nodes. An HPC cluster may be as simple as two networked identical compute nodes or as complex as thousands of nodes connected together via high speed network.
(3) *The management node*: this node is used for administrative purposes, such as installing, configuring and administering all other client nodes.
(4) *Network Interconnect*: currently, there are several network interconnects that provide low latency (less than 5 microseconds) and high bandwidth (multiple Gbps). The suitable HPC interconnect is determined by the application that is intended to run on the cluster. Two of the leading products are Myrinet [16] and Quadrics. More recently, InfiniBand [7] has entered the high performance computing market. Unlike Myrinet and Quadrics, InfiniBand

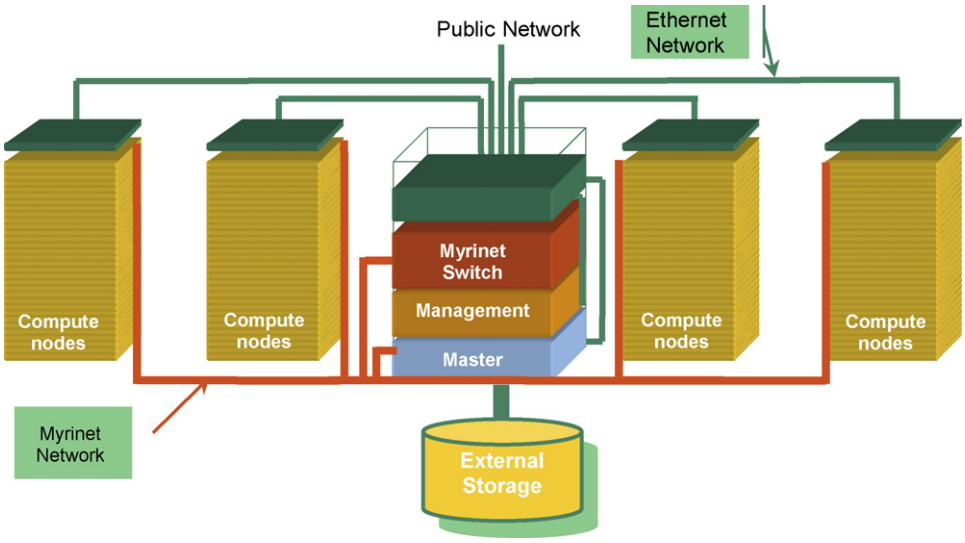

Fig. 1. A general "large-scale" Beowulf cluster layout.

Table 1
Network interconnects comparison [1,18]

	Gbit Ethernet	Myrinet D-cards	Quadrics QsNet	InfiniBand
Throughput	120 MB/s	473 MB/s	308 MB/s	841 MB/s[b]
Latency[a]	12.1	6.8	4.6	6.7
Price	Cheap	Moderate	Moderate-to-expensive	Expensive

[a] Measures are in microseconds.
[b] Limited by the PCI-X bus.

was initially introduced as a generic interconnect for inter-process communication and I/O [3]. Nevertheless, its high performance and scalability make it also attractive as a communication layer for high performance computing. Tests [1,2] show that for small MPI messages, InfiniBand outperform other interconnects due to the higher bandwidth, while Quadrics fits small MPI messages the best due to the very low latency. Table 1 shows each interconnect option with its peak latency and throughput measures.

In the next section, we start our proposed design by introducing redundancy and failover components in the lowest level of the HA–HPC clusters.

4. FT-HPC in the hardware layer

As we have seen from the standard Beowulf design, there are certain components where they are considered to be a single point of failure, such as the master node and the network switches. In earlier times, *cost* was a valid reason for systems designers to emphasize on the application needs and minimize the cost of building an HPC cluster by removing all the costly redundant hardware components. However, as the cost of commodity hardware is declining, combining HPC and HA architectures is becoming feasible to achieve HA cluster that is used for High Performance Computing. For instance, various techniques are currently available to have a redundant master node [3,12,13,15,18]. These implementations include active/active, active/hot-standby, and active/cold-standby nodes. In the active/active, both master nodes simultaneously provide services to external requests and once one node is down, the other will take over total control. Whereas, a hot-standby head node monitors system health and only takes over control when there is an outage at the primary node. The cold-standby architecture is very similar to the hot-standby, except that the backup head is activated from a cold start.

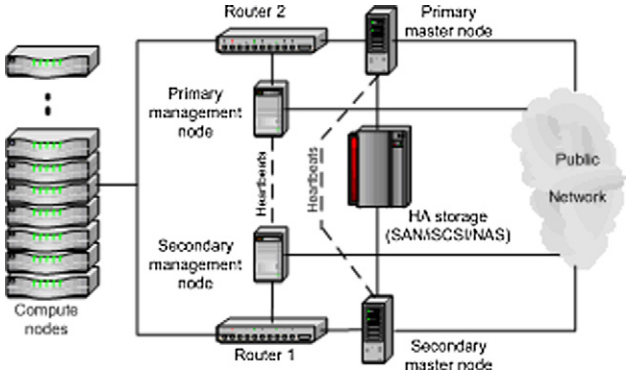

Fig. 2. A general fault-tolerant HPC cluster architecture.

Figure 2 shows a completely hardware-modified FT-Beowulf architecture. Each master node has two VLANs; one is connected to the Internet by a public VLAN, and the other NIC is connected to the private LAN. The standby primary server monitors the primary server and waits for taking over when a failure in the primary server is detected. Note that the Local LAN Switches are completely redundant and the cluster may be operational even if one switch fails.

5. FT-HPC in the management layer

Systems that have the ability to hot-swap hardware components need a management software layer that understands the concept of dynamic system configuration, in order to keep the system alive. One of these management applications is the High Availability Open Source Cluster Application Resource (HA-OSCAR). HA-OSCAR is an Open Source project that aims toward non-stop services in the HPC environment through a combined power of High Availability and Performance Computing solutions [2].

5.1. HA-OSCAR

The OSCAR project is built by a mixture of academic and research members including: Bald Guy Software (BGS), DELL, IBM, Intel, MSC.Software, Indiana University, the National Center for Supercomputing Applications (NCSA), Oak Ridge National Laboratory (ORNL), and University of Sherbrooke [1]. The HA-OSCAR project's primary goal is to enhance a Beowulf cluster system for mission-critical applications and sensitive HPC infrastructures. Its basic package includes a set of "core" toolkits needed to build and maintain a cluster [2]. Other included tools cover most, if not all, commonly used HPC applications, such as: LAM/MPI, PVM and MPICH for running parallel applications, Maui Portable Batch System (PBS) for batch job scheduling, OpenSSH for secure remote login, and System Installation Suite (SIS) [1].

HA-OSCAR addresses service level faults via the Adaptive Self Healing technique (ASH) [2,3]. ASH MON daemon monitors service(s) availability at every interval (default is 5 seconds) and triggers alerts upon failure detection. When a failure is triggered, a hot-standby node takes over. This hot-standby node is a clone of the active node that contains the entire HA-OSCAR software bundle and will process user requests when the active master node fails.

To see how the master node's hot-swapping process works with HA-OSCAR, consider the following example that is illustrated in Fig. 3: HA-OSCAR assigns a primary server public IP address to be used for external accesses and a private IP address to connect to the compute nodes (a private VLAN as users are not allowed to access the compute nodes). For the standby server, the public IP address is initially unassigned and its private IP address is configured as shown in Fig. 3.

When a primary server failure occurs, all its network interfaces will go down. HA-OSCAR will trigger the network disconnection and the standby server takes over and clones the primary server network configuration. The standby server will automatically mimic its both public and private network interfaces to be the same as the original IP addresses for the primary master node. This IP cloning process only takes 3–5 seconds [3].

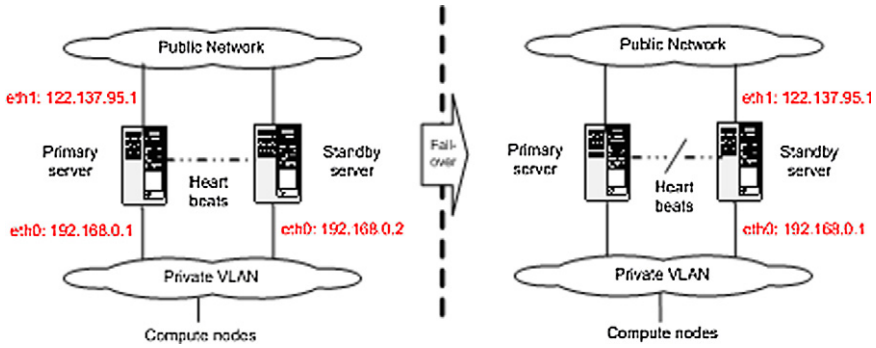

Fig. 3. HA-OSCAR failover process to the master node.

6. FT-HPC in the application layer

Up to this point, we have considered the hardware and management elements in designing a FT-HPC cluster. Now we present, in more details, the HA issues in the application level. Although other implementation exists such as PVM (Parallel Virtual Machine), we focus our study on the Message Passing Interface (MPI) implementations, mainly because it is becoming the de facto standard communication protocol for parallel processing and HPC, especially when performance is the main concern [1].

6.1. Introduction to MPI

MPI is a library of routines that can be called from any programming language (mainly C or FORTRAN), on a distributed memory system. It is designed to allow a network of heterogeneous machines to be used as a single distributed parallel processor. MPI's advantage over older message passing libraries is that it is both portable and fast (because each MPI implementation is optimized for the hardware it runs on) [1].

The need for a fault tolerant MPI standard arises from the growing concern with the reliability of processors, communication and systems' structure. As current HPC systems increase in size, fault tolerance in MPI becomes a very important concern for critical high performance applications using the MPI library. W. Gropp and E. Lusk [1] define the requirements of a fault-tolerance MPI as:

– Failure can be detected.
– Information needed to continue the computation is available.
– The computation can be restarted.

Moreover, they see that the highest level of MPI "survival" is that when MPI implementation automatically recovers from faults while the MPI program continues without significant change to its behavior. Their definition to the second level of survival is when the applications are notified of the problem and are prepared to handle it. In each of two cases, the application may proceed without restarting because the program has enough information about the failing process and can assign another running process to pick up the work from there. A different level of survival is that an application aborts and restarts from a checkpoint. Here, the states of all processes are saved outside the processes themselves, in a stable storage for example.

In the next section, we analyze some of the techniques that are used to build such fault tolerant MPI packages, namely: Checkpointing, Message Logging and the Worker/Manager algorithm. We compare these concepts and propose minor modifications to make them suit our complete HA–HPC design. Later, we review some of the existing fault tolerant MPI packages that are based on the mentioned techniques. Finally, we conclude our work by benchmarking a small-scale cluster and stating our findings and comments.

6.2. MPI error handlers

Before moving to the various fault tolerant MPI techniques, we first present the pre-defined MPI error-handlers and describe their routines.

MPI error-handlers specify the action to be taken when the MPI program runs through a failure. The specified error handling routine is used for any MPI exception that occurs during an MPI call for a communication with the communicator. The set of errors calls that are handled by MPI is implementation-dependent [1]. Therefore, a stable MPI implementation will attempt to handle as many errors as possible. Errors that are not handled by MPI will be either handled by the error handling mechanisms of the language run-time or the operating system [2].

The MPI error-handlers do not necessarily allow the user to continue to use MPI. Instead, they allow a user to issue user-defined error messages and to take actions unrelated to MPI (such as flushing I/O buffers, having a memory dump or calling other outside routines) before a program exits. However, it is not required from a non-fault tolerant MPI implementation to use error handlers [5,6,8].

MPI error-handlers can be either built in or user defined [1]. The built-in error-handlers are MPI_ERRORS_ARE_FATAL (The default error-handler) and MPI_ERRORS_RETURN. The first error-handler indicates that if an MPI function returns unsuccessfully then all the processes in the communicator will abort. The latter handler indicates that MPI functions will attempt to return an error code. User defined error-handlers are attached to the MPI communicators. The ability to support user-defined error-handlers is important for developers when building their own MPI interfaces.

6.3. Fault tolerant MPI techniques

In this section, we cover the commonly used techniques for implementing fault tolerant MPI. In particular, we study checkpointing (Global, independent and coordinated checkpointing), Massage Logging, and Manager/Worker techniques [8,10,14,16].

6.3.1. MPI checkpointing

Checkpointing is a technique where the status of the computation is saved in different stages of running, allowing the application to be restarted from that point in the event of a failure. The primary advantage of checkpointing is that it is easy to implement. However, it is often considered expensive because the time taken to do a checkpoint can rapidly grow. Therefore, fast I/O media is usually used to connect the cluster's nodes to the shared storage (e.g. fiber links), and checkpointed data are saved in a reliable storage that should not be affected by the application failure. Normally, a parallel file system, such as GPFS [3] or SNFS [2], is used for checkpointing, since more than one process will be writing to the same storage.

Checkpointing can be classified as user-directed and system-directed. In user-directed checkpointing, the programmer manually writes out any data that will be needed to restart the application. The user has to ensure that all needed data is saved, which might not be an easy task. Moreover, the checkpoints must be taken at particular points in the program, typically when there is no interaction between processes, which again can be difficult, especially for programs that are not well-structured. Certain tools and APIs exist to assist programmers to determine where and when to checkpoint [3]. Although user-directed checkpointing seems complicated, system-directed checkpointing is much harder to implement because the processes states might be scattered throughout the cluster [2].

6.3.1.1. Locally-stored global checkpointing Locally-stored global checkpointing consists of taking a consistent snapshot of the system at a time. A snapshot is a collection of checkpointing images (one per process) with each channel state. "Locally-stored" refers to the use of a local cache on each node instead of having a shared repository for all processes, while the term "global" refers to the synchronization between processes to take the snapshot.

In locally-stored global checkpointing, every process makes a copy of the checkpoint image in a local repository. When a restart occurs, instead of collecting their last checkpoint image from the checkpoint server, processes get their data them from their local repository [1]. However, having a local repository for each process introduces another complication: since the processes do not share a single repository when a restart takes place, care has to be taken to ensure the consistency of the global image. To assure consistency, the checkpoint-scheduler gets notified and then validates the global view once a process has successfully checkpointed a state. Another problem with distributed local repositories is that they independently need to be cleaned from time to time.

6.3.1.2. Independent checkpointing In independent checkpointing technique, processes are not synchronized to do checkpoints but each process does it autonomously, and checkpointed data are usually stored in a centralized place. Recovery requirements for independent checkpointing are similar to the locally-stored global checkpointing in a way that dependencies should be recorded by the checkpoint-scheduler so that processes can jointly roll back to a consistent global state.

The complexity of synchronizing and achieving a global consistent state by each process is fairly complex and does not justify the need for independent or locally-stored checkpointing. For these reasons, coordinated checkpointing is becoming very popular [1].

6.3.1.3. Coordinated checkpointing In coordinated checkpointing, all processes synchronize to write their state to a global repository all at once. It is considered a better algorithm than locally-stored checkpointing because it is simpler and that the saved state is automatically globally consistent [1]. Moreover, it avoids the cost of cleaning the scattered local repositories from time to time.

Now we see how coordinated checkpointing is applied to build a fault tolerant MPI. One form of coordinated checkpointing is the use of two-phase blocking protocol, as demonstrated by [1]. In this model, a coordinator node sends a CHECKPOINT_REQUEST MPI message to all processes. When received, all running processes do a checkpoint in the global repository, queue any subsequent message handed to it by the application it is executing, and acknowledge to the coordinator that the checkpoint is completed. When the coordinator receives an acknowledgment from all processes, it multicasts a CHECKPOINT_DONE message back to continue the normal operation. It is important to mention that when a process receives an MPI message after receiving the checkpoint request from the coordinator, will considers it to be part of the local checkpoint. At the same time, outgoing messages are queued locally until the CHECKPOINT_DONE message is received.

A major drawback of the coordinated checkpointing technique is the restart time after a single crash. This long time is due to the overhead made by the checkpoint servers when they dump or restore their snapshots from the shared storage at once. To solve this deficiency, the coordinator may multicast a CHECKPOINT_REQUEST only to these processes that they participate on the recovery of the coordinator. Then, the receiver checks if it has communicated to other processes before receiving the message from the coordinator. If so, it forwards the checkpoint request to this process because it is considered indirectly-related to the coordinator. When all processes are identified, a CHECKPOINT_DONE message is sent to let the processes continue the normal operation.

Table 2 shows a summarized comparison between the discussed checkpointing techniques, based on the checkpoint and restart mechanism, the storage configuration and the added complexity.

6.3.2. Message logging

As checkpointing can grow rapidly and become very expensive, message logging is developed in order to reduce the checkpointing cost, but still enable recovery. The basic idea underlying message logging is to log all the operations of message transmissions, and replay them in case of failures to reach a globally consistent state, without having to restore that state from stable storage. In initial checkpoint state is needed as a starting point, and then all messages that have been sent since are simply replayed from the log and retransmitted accordingly.

Message logging technique has the advantage of avoiding to checkpoint the whole application, and therefore save storage space. However, it is clear that message logging technique has to coexist with checkpointing. That is, a starting checkpoint should be available to implement message logging. Therefore, checkpointing performance affects the overall message logging performance as well.

Table 2
Checkpointing techniques comparison

	Locally-stored global checkpointing	**Independent checkpointing**	**Coordinated checkpointing**
Checkpoint	All at once	Independent of each other	All at once
Restart	All at once	All at once	Subset of processes
Storage	Local to each process	Centralized	Centralized
Complexity	Easy to checkpoint, difficult to restart	Difficult to checkpoint and restart	Easy to checkpoint and restart

6.3.3. The manager/worker technique

In the generic architecture, the MPI process does not connect directly to the other ones. Instead, it uses a communication daemon that handles sending, receiving, recording messages, and establishing connections with all components of the system [1]. In the standard MPI, the failure of any one MPI process affects all processes in the communicator, even those that are not in direct communication with the failed process [1]. In contrast, in non-MPI client–server programs, the failure of a client does not effect on the server, because each peer is aware about the status of the other peer. For example, the client can easily recognize that the server has failed and stop communicating with it. W. Gropp and E. Lusk [1] tried to mimic this scheme and apply it in the MPI context. In this algorithm, a manager process use MPI_Comm_spawn to create the workers and submits small tasks to them, while keeping track of them. Then, workers return their results to the manager, simultaneously requesting a new task. This sort of communication makes it easy to recover after faults because the manager keeps a copy of the task specification and can simply re-assign it to another worker if one dies. In particular, the manager marks the failing communicator as invalid and does not use it again. When a worker dies, the MPI_Comm_spawn routine is used to replace the worker and continue processing with no fewer workers. According to [1], this program may not work on every implementation of MPI, because the MPI implementation must be able to return a non-success return code in the case of a communication failure such as an aborted process or failed network link [1].

A simple modification to the worker/manager technique in order to make it work is to add a timeout period in which the manager assumes the death of the worker process. This way, the manager process is notified about the dieing process (or worker) and assigns the task again to another worker. The timeout period can be a user variable and it all depends on the application that runs on top of the MPI layer. If performance is desired, then this timeout period can be minimal. On the other hand, the timeout period can be extended if FT is the main concern. Moreover, we may reduce the amount of messages exchanged by having the manager and all the workers write their status into a single shared storage, using a reliable parallel file system [2,3]. This way, when a worker fails and is assigned as a bad node, the manager allocates a new stand-by node, which reads the last status of the failing node and picks up the work from there.

6.4. Related work on fault tolerant MPI

Many fault tolerant MPI implementations exist, such as LAM/MPI, Open MPI, WMPI (Windows implementation), and FT-MPI, etc. The main difference between these implementations is the way they react to process or nodes failures in a way beyond that of the standard MPI interface. In particular, several implementations direct their fault tolerant techniques to the application level, while other techniques target their implementation to the transport and data-link levels [2]. Now we study two of these implementations in more detail.

6.4.1. StarFish MPI

The initial implementation of StarFish runs on Linux and supports both Myrinet and Ethernet communication links [2]. Each node in a Starfish cluster runs a daemon, and all Starfish daemons form a process group. Starfish daemons maintain some data for each application process running on the machine, as well as some shared state that defines the current cluster configuration and settings. These daemons are responsible for interacting with clients, spawning the application processes, tracking and recovering from failures, and maintaining the system configuration [2]. Further, each application process is composed of 5 major components. These are: a group handler, which is responsible for communicating with the daemon, an application part, which includes the user supplied MPI code, a checkpoint/restart module, an MPI module, and a virtual network interface. These components communicate using an object bus based listener model [2]. To guarantee low latency and minimal impact on performance, the application part has a separate fast data path to and from the MPI module.

Starfish offers two forms of fault-tolerance for applications. The main fault-tolerant mechanism employed by Starfish is checkpoint/restart. The checkpoint/restart module of Starfish is capable of performing both coordinated and uncoordinated checkpoint, which is either system driven or application driven [3]. Thus, when a node failure occurs, Starfish can automatically restart the application from the last checkpoint. The other form of fault tolerance offered by Starfish is more application dependent, and is suitable mostly to applications that can be trivially parallelized. For such applications, whenever a node that runs one of the application processes crashes, the event is delivered to all

surviving application processes. Once the surviving members find out about the failure of a node, they repartition the data sets on which each process computes, and continue to run without interruption.

When an application is submitted to Starfish, the client determines the fault tolerant policy that should be applied to this application, i.e., should automatic restart or process notifications be used, and some rules regarding how to choose the node on which a process will be started after a partial failure.

6.4.2. FT-MPI

FT-MPI is a fault tolerant MPI implementation that changes the semantics of the original MPI to allow the application to tolerate process failure [17,20–22]. In particular, FT-MPI can tolerate the failure of $n - 1$ processes in an n-process job, provided that the application recovers the data structures and the data of the failed processes [22].

Handling failures in FT-MPI consists of three steps: the first two are failure detection and notification. In these two steps, the run-time environment discovers failures and all remaining processes in the parallel job are notified about them. The third step is recovery, which consists of recovering the MPI library, the run-time environment, and the application.

There are two modes that can be specified when running an FT-MPI application, these are [22]:

(1) *The communicator mode*: this mode indicates the status of an MPI object after recovery. FT-MPI offers four different communicator modes that can be specified when starting the application:

(a) ABORT: this mode makes the application abort when an error occurs.
(b) BLANK: in this mode, failed processes are not replaced and all surviving processes have the same rank as before the crash.
(c) SHRINK: in this mode, failed processes are not replaced. However, processes might have a new rank after recovery.
(d) REBUILD: this is the default mode in FT-MPI. Failed processes are re-spawned, surviving processes have the same rank as before.

(2) *The* "*communication mode*": this mode indicates how to treat the messages that are on the way while an error occurs. FT-MPI provides two different communication modes for this situation:

(a) CONT/CONTINUE: in this mode, all operations which returned the error code MPI_SUCCESS will finish successfully, even if a process failure occurs during the operation.
(b) NOOP/RESET: in this mode, all ongoing messages are dropped and the application returns to its last consistent state. All currently ongoing messages are ignored.

6.4.3. Other fault tolerant message passing implementations

As mentioned previously, MPI has a rich set of communication functions, which makes MPI favored over other implementations [2]. However, there are other popular parallel interfaces, such as PVM (Parallel Virtual Machine), and its various fault tolerant implementations, such as DynamicPVM and MPVM that provide the same MPI functionality. PVM is different than MPI in a way that it is built around the concept of a virtual machine, so it has the advantage when the application is going to run over a networked collection of hosts, especially if the hosts are heterogeneous. Moreover, PVM contains resource management and process control functions that are important for creating portable applications that run on clusters of workstations. G. Geist, J.A. Kohl, P.M. Papadopoulos in [1] explore more on the differences between PVM and MPI.

For completeness, we view one of PVM implementations and study how it handles fault tolerant in parallel applications.

6.4.3.1. DynamicPVM In general, PVM transmit communication messages using daemons, i.e. a message is first transferred to the sender's daemon, then forwarded to the daemon of the receiver and then delivered to the receiver [3].

While standard PVM offers only a static process assignment to the application, DynamicPVM provides dynamic process assignment and task scheduling, so that processes are migrated during runtime during failures. In particular, when a process failure is triggered in DynamicPVM, the local daemon on a new node prepares itself to receive the messages from the failing node, and sets its message buffer. The routing information of the local daemon on the old

node gets updated so that messages which are still being sent to the old node are forwarded to the daemon on the new node. The sender daemon is informed about the new location of the process so that, in future, it sends directly to this process. One limitation in the current DynamicPVM implementation is that it is only possible to migrate one process at a time [1].

7. Benchmarking and performance results

In this section, we analyze our small-scale HA–HPC functionality and evaluate its performance. In our evaluation, we used eight-clustered nodes, each equipped with a Pentium 4 3.2 GHz processor, 1.0 GHz memory and run Linux RedHat 9.0. Two of these nodes are set as the master nodes, a primary and a hot-standby, and are managed by HA-OSCAR. The other six nodes are configured as the compute nodes and are interconnected by a 100 MB/s network. We used DynamicPVM and FT-MPI interfaces to handle our parallel jobs, with POV-Ray package [1] on top. POV-Ray is a 3D ray-tracing engine that takes users' input and simulates the way light interacts with the defined objects to create 3D pictures and animation. POV-Ray has the ability to distribute a rendering across multiple systems, in the form of one-master and many-slave task. The master node has the responsibility of dividing the image up into small blocks, which are assigned to the slaves. When the slaves finish rendering the blocks, they are sent back to the master, which combines them to form the final image. In our experiment, we used a 2600 bytes script file to render a 1024×768 picture with an output size of 2.3 MB.

First, we ran our benchmarks with varying number of nodes, while keeping the number of processes in each node fixed, as shown in Fig. 4. Clearly, the best performance/nodes was obtained while rendering on four nodes. This performance increase rate has dropped as we increased the nodes to eight. We expect that this performance gain would be flattened with 12 nodes.

Next, we ran other benchmarks while varying the number of nodes and the running processes at the same time, as in Fig. 5. The best run cases were when the number of processes matched the number of nodes that are running (i.e. one process on each node). By increasing the number of processes beyond the number of nodes, the inter-process communication overhead became noticeable, causing the image to take more time to render.

Figure 6 shows how much TCP traffic is generated on the master node. Obviously, the traffic increases as we increase the number of nodes, due to the further processes' distribution among nodes. In Fig. 7, we show how image resolution affects the rendering time. While the dashed curve demonstrates the rendering time when we enforce POV-

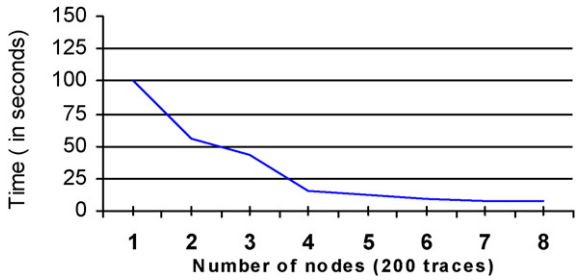

Fig. 4. Number of nodes vs. rendering time.

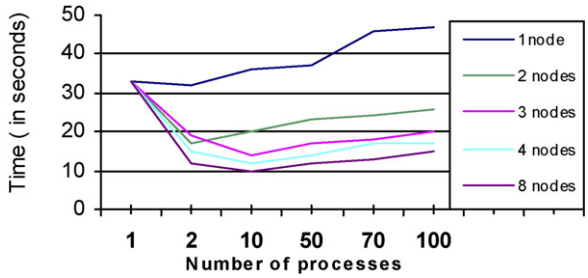

Fig. 5. Number of processes vs. rendering time.

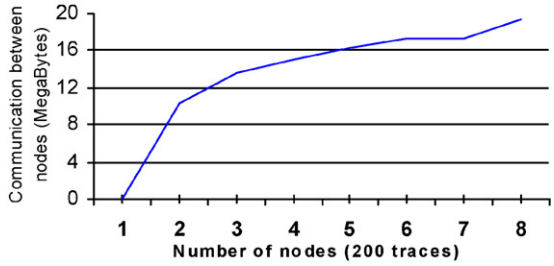

Fig. 6. Communication traffic node on the master.

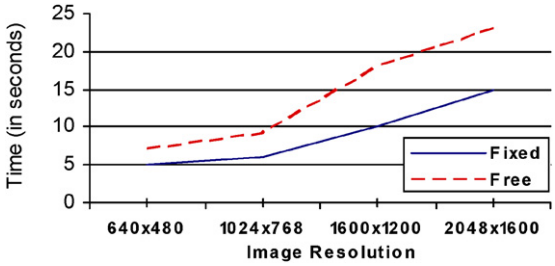

Fig. 7. Image resolution vs. rendering time.

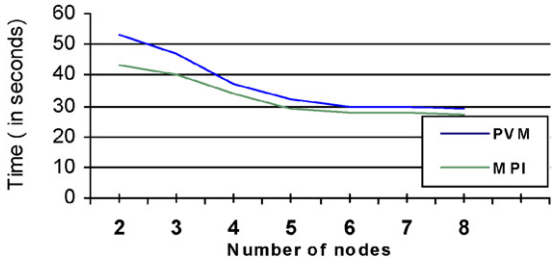

Fig. 8. DynamicPVM vs. FT-MPI.

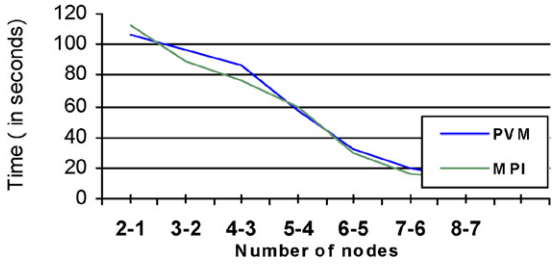

Fig. 9. DynamicPVM vs. FT-MPI (with faults).

Ray to evenly distribute the processes among the nodes, the continuous curve shows the rendering time when we let the program freely distribute the processes among nodes.

Figure 8 shows the performance comparison between FT-MPI and DynamicPVM when rendering the same image on both message passing interfaces, using 200 processes in each node. FT-MPI showed a performance increase compared to DynamicPVM, especially with low number of nodes, but they performed comparatively with 8 nodes. Finally, we tested FT-MPI and DynamicPVM performance with a single node fault. Figure 9 shows the time taken to render the image while dropping one node during the process (for example, 2–1 in the figure means dropping from 2 nodes to one). Both interfaces showed approximately the same performance, even though they use different recovery schemes, as we discussed in Section 2.

8. Conclusion

One of the challenges in a High Performance Clustering is to minimize system failures, maintain maximum performance and provide the highest possible level of system availability. However, these goals are not easy to achieve, due to the various issues that have do be managed. These include maintaining high availability at the hardware layer, cluster manageability and fault tolerance in the application level.

In this paper, we demonstrated in detail the architecture of building a FT-HPC cluster, considering all aspects. In particular, we explored the hardware and the management layers of the HA–HPC cluster design, as well as a more focused study on the parallel-applications layer (i.e. fault tolerant MPI implementations). We also showed that the Manager/Worker algorithm that was proposed by [1] can be improved to suit most of the FT-MPI implementations.

Finally, we put together a small-scale fault tolerant HPC cluster using HA-OSCAR and different MPI implementations, to study the behavior of such a system. Our results show that combining HPC and HA architectures is feasible, in order achieve HA cluster that is used for High Performance Computing.

References

[1] F. Pister, L. Hess, V. Lindenstruth, Fault tolerant grid and cluster systems, Kirchhoff Institute of Physics (KIP), University Heidelberg, Germany, 2004, pp. 360–363.
[2] G. Fagg, J. Dongarra, Building and using a fault tolerant MPI implementation, Int. J. High Perform. Comput. Appl. 18 (3) (2004) 353–361.
[3] I. Haddad, C. Leangsuksun, R. Libby, T. Liu, Y. Liu, S. Scott, Highly reliable Linux HPC clusters: Self-awareness approach, in: Proceedings of the 2nd International Symposium on Parallel and Distributed Processing and Applications, Hong Kong, China, December, 2004.
[4] J. Mugler, T. Naugthon, S. Scott, C. Leangsuksun, OSCAR clusters, in: Proceedings of the Linux Symposium, Ottawa, Canada, 2003.
[5] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, F. Cappello, Improved message logging versus improved coordinated checkpointing for fault tolerant MPI, in: Proceedings of the 6th International Conference on Cluster Computing, CLUSTER'04, San Diego, CA, USA, 2004.
[6] G.E. Fagg, J.J. Dongarra, FT-MPI: Fault tolerant MPI, in: Supporting Dynamic Applications in a Dynamic World, PVM/MPI, 2000, pp. 346–353.
[7] G.E. Fagg, J.J. Dongarra, Building and using fault-tolerant MPI implementation, Int. J. High Perform. Comput. Appl. 18 (2004) 353–361.
[8] G. Stellner, 1996. CoCheck: Checkpointing and process migration for MPI, in: Proceedings of IPPS'96, The 10th International Parallel Processing Symposium, Honolulu, HI, USA, 1996, pp. 526–531.
[9] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, J. Dongarra, MPI: The Complete Reference, vol. 1, second ed., MIT Press, Cambridge MA, 1998.
[10] K. Li, J.F. Naughton, J.S. Plank, Low-latency, Concurrent checkpointing for parallel programs, IEEE Trans. Parallel Distrib. Systems (1998) 874–879.
[11] William Gropp, Ewing Lusk, Fault tolerance in MPI programs, in: Proceedings of the Cluster Computing and Grid Systems Conference, December 2002.
[12] A. Gidenstam, B. Koldehofe, M. Papatriantafilou, P. Tsigas, Dynamic and fault-tolerant cluster management, Technical report 2005-10, Computer Science and Engineering, Chalmers University of Technology, April 2005.
[13] A. Hasegawa, Hiroshi Matsouka, K. Nakanishi, Clustering software for Linux-based HPC, NEC Res. Dev. High Perform. Comput. 44 (2003) 60–63.
[14] B. Polgar, Designing the reconfiguration strategies of fault tolerant servers, in: The Third European Dependable Computing Conference, Czech Republic, September, Prague, 1999.
[15] P. Sobe, Fault-tolerant Web services on a computer cluster, in: Dependable Computing—EDCC-3, The Third European Dependable Computing Conference, Czech Technical University in Prague, Prague, 1999.
[16] F. Sultan, A. Bohra, S. Smaldone, Y. Pan, P. Gallard, I. Neamtiu, L. Iftode, Recovering Internet service sessions from operating system failures, IEEE Internet Comput. 9 (2005) 17–27.
[17] R. Aulwes, D. Daniel, et al., Architecture of LA-MPI, a network-fault-tolerant MPI, in: Proceedings of the 18th International Parallel and Distributed Processing Symposium, 2004, pp. 26–30.
[18] A. Azagury, D. Dolev, et al., Highly available cluster: A case study. Fault-tolerant computing, in: 24th International Symposium on Fault Tolerant Computing System, TX, USA, 1994, pp. 404–413.
[19] A. Agbaria, R. Friedman, Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations, in: 8th International Symposium on High Performance Distributed Computing, HPDC-8'99, IEEE Computer Soc. Press, August 1999.
[20] S. Rao, L. Alvisi, H. Vin, Egida: An extensible toolkit for low-overhead fault-tolerance, 29th International Fault-Tolerant Computing Symposium, Los Alamitos, CA, 1999, pp. 48–55.
[21] G. Stellner, CoCheck: Checkpointing and process migration for MPI, in: 10th International Parallel Processing Symposium, IEEE Computer Soc. Press, 2004, pp. 526–531.
[22] Z. Chen, G. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, J. Dongarra, Fault tolerant high performance computing by a coding approach, in: Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, IL, USA, 2005, pp. 15–17.