A Job Pause Service under LAM/MPI+BLCR for Transparent Fault Tolerance *

Chao Wang[1], Frank Mueller[1], Christian Engelmann[2], Stephen L. Scott[2]

[1] Department of Computer Science, North Carolina State University Raleigh, NC
[2] Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN
mueller@cs.ncsu.edu, phone: +1.919.515.7889, fax: +1.919.515.7896

Abstract

Checkpoint/restart (C/R) has become a requirement for long-running jobs in large-scale clusters due to a mean-time-to-failure (MTTF) in the order of hours. After a failure, C/R mechanisms generally require a complete restart of an MPI job from the last checkpoint. A complete restart, however, is unnecessary since all but one node are typically still alive. Furthermore, a restart may result in lengthy job requeuing even though the original job had not exceeded its time quantum.

In this paper, we overcome these shortcomings. Instead of job restart, we have developed a transparent mechanism for job pause within LAM/MPI+BLCR. This mechanism allows live nodes to remain active and roll back to the last checkpoint while failed nodes are dynamically replaced by spares before resuming from the last checkpoint. Our methodology includes LAM/MPI enhancements in support of scalable group communication with fluctuating number of nodes, reuse of network connections, transparent coordinated checkpoint scheduling and a BLCR enhancement for job pause. Experiments in a cluster with the NAS Parallel Benchmark suite show that our overhead for job pause is comparable to that of a complete job restart. A minimal overhead of 5.6% is only incurred in case migration takes place while the regular checkpoint overhead remains unchanged. Yet, our approach alleviates the need to reboot the LAM run-time environment, which accounts for considerable overhead resulting in net savings of our scheme in the experiments. Our solution further provides full transparency and automation with the additional benefit of reusing existing resources. Executing continues after failures within the scheduled job, i.e., the application staging overhead is not incurred again in contrast to a restart. Our scheme offers additional potential for savings through incremental checkpointing and proactive diskless live migration, which we are currently working on.

1 Introduction

Large-scale clusters with thousands of nodes often experience a mean-time-to-failure (MTTF) in the order of tens of hours. For example, BlueGene/L (BG/L) at Livermore National Laboratory with a total of 65,536 nodes was experiencing faults at a rate of 48 hours during initial deployment [16]. These faults occurred at the level of a dual-processor compute card but implied that a 1024-processor midplane had to be temporarily shut down to replace the card. As another example, the work by Philp [21] extrapolates the mean time between failure (MTBF) for petaflop machines to be just 1.25 hours based on current system technology.

To prevent valuable computation to be lost due to failures, checkpoint/restart (C/R) has become a requirement for long-running jobs. Current C/R mechanisms commonly allow checkpoints to be written to a global file system so that in case of failure the entire MPI (Message Passing Interface) job can be restarted from the last checkpoint. One example of such a solution is LAM (Local Area Multicomputer)/MPI's C/R support [24] through Berkeley Labs C/R (BLCR) [12]. A complete restart, however, is unnecessary since all but one node are typically still alive.

The contribution of this paper is to avoid a complete restart and retain execution of MPI jobs as nodes fail, as depicted in Figure 1. As such, we pause the MPI processes on live nodes and migrate the MPI processes of failed nodes onto spare nodes. We have developed a transparent mechanism as an extension to LAM/MPI+BLCR that reuses operational nodes within an MPI job. Functional nodes are rolled back to the last checkpoint, retaining internal communication links within LAM/MPI+BLCR, before the corresponding processes are paused. Meanwhile, a failed node is replaced with a spare node where the corresponding MPI task is recovered from the last checkpoint. Hence, live nodes remain active while failed nodes are dynamically and transparently replaced. This solution removes any requeuing overhead by reuses existing resources in a seamless and transparent manner.

Our solution comprises several areas of innovation within LAM/MPI and BLCR: (1) *crtcp*, one of the Request Progression Interface (RPI) options [25] of LAM/MPI, is enhanced to reuse the network connections between live nodes upon the faults. (2) Lamd (LAM daemon) is complemented with a scalable group communication framework based on our prior work [29], which notifies the Lamd of live nodes about replacement nodes. (3) Lamd is supplemented with a novel scheduler that transparently controls periodic checkpointing and triggers migration upon node

*This work was supported in part by NSF grants CCR-0237570 (CAREER), CNS-0410203, CCF-0429653 and DOE DE-FG02-05ER25664. The research at ORNL was supported by the Mathematics, Information and Computational Sciences Office, Office of Advanced Scientific Computing Research, and DOE DE-AC05-00OR22725 with UT-Battelle, LLC.

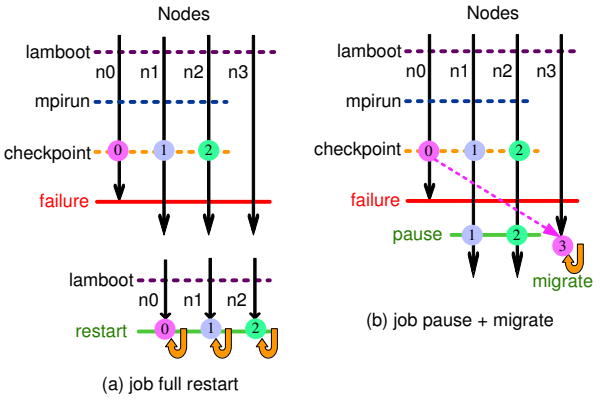

Figure 1. Pause & Migrate vs. Full Restart

failures while ensuring that live nodes are paused, which prevents LAM/MPI from prematurely terminating a job. (4) BLCR is supplemented with a job pause/restart mechanism, which supports multi-threading.

We have conducted a set of experiments on a 16-node dual-processor (each dual core) Opteron cluster. We assess the viability of our approach using the NAS (NASA Advanced Supercomputing) Parallel Benchmark suite. Experimental results show that the overhead of our job pause mechanism is comparable to that of a complete job restart, albeit at the added benefits of full transparency and automation combined with the reuse of existing resources in our case. Hence, our approach avoids requeuing overhead by letting the scheduled job tolerate the fault so that it can continue to execute. We are furthermore investigating additional benefits of our scheme for incremental checkpointing and proactive diskless live migration.

The paper is structured as follows. Section 2 presents the design of our transparent fault-tolerance mechanism. Section 3 identifies and describes the implementation details. Subsequently, the experimental framework is detailed and measurements for our experiments are presented in Section 4 and 5, respectively. Finally, the contributions are contrasted with prior work in Section 6, and the work is summarized in Section 7.

2 Design

This section presents an overview of the design of transparent fault tolerance with LAM/MPI+BLCR. The approach extends the checkpoint/restart framework of LAM/MPI with an integrated group communication framework and a fault detector, an internal schedule mechanism (Figure 3), the job pause mechanism and actual process migration (Figure 2). As a result, BLCR is supplemented with the new capability of *cr_pause*, the transparent job-pause functionality.

In the following, the design of the protocol is given. As depicted in Figure 3, the scheduler daemon acts as a coordination point between the *membership daemon* and *mpirun*, the initial LAM/MPI process at job invocation. The main

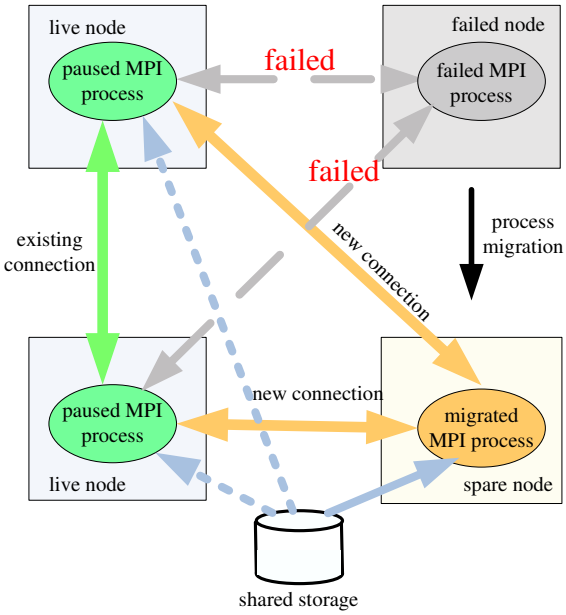

Figure 2. Job Pause and Migrate Mechanism

logical steps can be summarized as follows. 1) The membership daemon monitors the system and notifies the scheduler daemon upon the faults. 2) The scheduler daemon coordinates the job pause for functional nodes and process migration for failed ones. 3) The nodes perform the actual pause and migration work, as depicted in Figure 2. 4) All active processes (MPI tasks) continue the MPI job.

1. **membership daemon** and **scheduler daemon**: initialization through *lamboot*;
2. **membership daemon**: maintains the group membership information and monitors for faults;
3. **membership daemon**: notifies the *scheduler* upon detecting a fault;
4. **scheduler daemon**: selects replacement nodes for the failed ones and notifies the respective *mpirun* processes governed by the LAM runtime environment;
5. **each notified mpirun**: propagates the job-pause request to each MPI process on live nodes and sends a process migration command to the replacement nodes;
6. **each MPI process on live nodes**: engages in job-pause; meanwhile,
7. **replacement nodes**: restart from the checkpoint file to migrate the process;
8. **each MPI process on live nodes/replacement nodes (migrated)**: continues/resumes execution.

The implementation details of this algorithm with respect to LAM/MPI are given in the next section.

2.1 Group Communication Framework and Fault Detector

In our prior work [29], we devised a scalable approach to reconfigure the communication infrastructure after node failures within the runtime system of the communication

layer. A decentralized protocol maintains membership of nodes in the presence of faults. Instead of seconds for reconfiguration, our protocol shows overheads in the order of hundreds of microseconds and single-digit milliseconds over MPI on BG/L with up to 1024 processors.

We complemented lamd, the low-level communication daemon of LAM/MPI running on each node, with this scalable group communication framework from our prior work. We further designed a fault detector based on a single timeout mechanism. Excessive delay in response from any process to a message request is assumed to indicate process (node) failure. In our framework, link failures are handled similarly to node failures since the cause of a long communication delay is not distinguished, *i.e.*, different causes of failure can be handled uniformly (and will be uniformly referred to as "node failures" or simply "failures"). When the detector determines a failure, it triggers the scheduler through a message containing the ID of the failed node.

2.2 Internal Schedule Mechanism in LAM/MPI

We next describe the mechanism to control a job's schedule upon node failure. This schedule mechanism is rooted within the LAM runtime environment, independent of the system-level batch scheduler governing job submission, such as Torque/OpenPBS (Portable Batch System). The protocol for the internal scheduler is as follows: It

- launches periodic checkpoint commands at a user-specified frequency;
- determines replacement nodes for failed nodes when a fault occurs; and
- launches job pause commands to *mpirun*s.

The mechanism has been implemented under the following design principles: The scheduler

- is integrated into the run-time environment;
- is decentralized, without any single point of failure; and
- provides good scalability due to the underlying group communication framework and its own functional layer on top.

The scheduler daemon is the main part of the schedule mechanism. In addition, *mpirun* also ties in to the schedule framework, most notably through the propagation of job pause commands.

In our current implementation, the schedule mechanism stores the information about the MPI job (such as *mpirun-pid@node-id*, *mpi-process-pid@node-id*, etc.) on a reliable shared storage or, in an alternative design, keeps them in memory. Both approaches have pros and cons. Nonetheless, the overhead of maintaining and utilizing such information is in the order of microsecond, regardless of the storage location. Hence, compared to overhead for C/R of up to tens of seconds, this bookkeeping overhead of the scheduler is insignificant. In our implementation, *mpirun* is responsible for logging the information about the MPI job on shared storage or in memory after a successful launch of the job. During finalization of the job, *mpirun* will disassociate this information.

2.3 Job-pause

As depicted in Figure 2, upon node failure, the job-pause mechanism allows the processes on live nodes to remain active by rolling back computation to the last checkpoint (rather than necessitating the traditional complete cold restart of an MPI job under LAM/MPI, which would incur long wait times in the job submission queues). At the LAM level, job-pause reuses the existing connections among the processes. Furthermore, at the C/R level (using BLCR), part of the state of the process is restored from the checkpoint state instead of a complete restart of the process. Currently, we use the existing process and its threads without forking or cloning new ones. Furthermore, we do not need to restore the parent/child relationships (in contrast to *cr_restart*), but we still restore shared resource information (*e.g.*, mmaps and files). LAM uses the module of *crtcp* to maintain the TCP connections (sockets) among the MPI processes.

BLCR restores the state of the MPI processes (*i.e.*, the sockets are kept open by the *crtcp* module during the *cr_pause*). Hence, we can safely reuse the existing connections among the processes. Even though BLCR does not support transparent C/R of socket connections, this does not adversely affect us here since LAM/MPI relies on communication *via crtcp* rather than through lower-level network sockets.

A pause of the MPI job process is initiated by *mpirun*. In response, the following sequence of events occurs:

1. **mpirun**: propagates the job-pause request to each MPI process on live nodes;
2. **BLCR on live nodes**: invokes the *cr_pause* mechanism;
3. **paused process**: waits for *mpirun* to supply the information about the migrated process;
4. **mpirun**: updates the global list with information about the migrated process and broadcasts it to all processes;
5. **paused process**: receives information about the migrated process from *mpirun*;
6. **paused process**: builds its communication channels with the migrated process;
7. **paused process**: resumes execution from the restored state.

Similar to the *checkpoint* and *restart* functionality of BLCR, the novel *pause* mechanism of BLCR also interacts with LAM through a threaded callback function. The callback is provided as part of our LAM enhancements and registered at the initialization of *crtcp*. The *pause* mechanism performs an ioctl() call to enter the pause state. As part of the pause functionality, a process rolls back its state to that of a former checkpoint, typically stored on disk. This rollback is the inverse of checkpointing, *i.e.*, it restarts a

process at the saved state. However, there are consequential differences. Pause/rollback reuses the existing process without forking a new one. Furthermore, existing threads in the process are reused. Only if insufficient threads exist will additional ones be created (cloned, in Linux terms). Hence, it becomes unnecessary to restore the Process ID (PID) information or re-create parent/child relationships from the checkpoint data.

2.4 Process Migration

When the scheduler daemon receives a node failure message from the membership daemon, it performs a migration to transfer processes, both at application and runtime level, from the failed node to the replacement nodes. This happens in a coordinated fashion between the *mpirun* processes and new processes launched at the replacement nodes,

Several issues need to be solved here: First, *mpirun* launches a *cr_restart* command on appropriate nodes with the relevant checkpoint image files. In our system, the scheduler daemon determines the most lightly loaded node as a migration target, renames the checkpoint file to reflect the change and then notifies *mpirun* to launch *cr_restart* from the relevant node with the right checkpoint file.

Second, the checkpoint files of all processes have to be accessible for replacement nodes in the system. This ensures that, at the fault time, the process can be migrated to any node in the system using the checkpoint file. We assume a shared storage infrastructure for this purpose.

Finally, knowledge about the new location of the migrated process has to be communicated to all other processes in the application. Since we operate within the LAM runtime environment, a node ID (instead of a node's IP address) is used for addressing information. Thus, migration within the LAM runtime environment becomes transparent, independent of external system protocols, such as Network Address Translation (NAT), firewalls, etc. Our system also updates the addressing information on-the-fly instead of scanning and updating all the checkpoint files, thereby avoiding additional disk access overhead for writes.

At the point of process migration, the following sequence of events occurs:

1. **mpirun**: sends a process migration command (*cr_restart*) to the replacement node;
2. **BLCR on the replacement node**: executes the *cr_restart* mechanism referencing the checkpoint file on shared storage;
3. **restarted process**: sends its new process information to *mpirun*;
4. **mpirun**: updates the global list with information about the migrated process and broadcasts it to all processes;
5. **restarted process**: builds its communication channels with all the other processes;
6. **restarted process**: resumes execution from the saved state.

Since all the information is updated at run-time, the normal *restart* operation of BLCR is executed from the replacement node without any modification.

3 Implementation Details

Our fault tolerance architecture is currently implemented with LAM/MPI and BLCR. Our components are implemented as separate modules to facilitate their integration into the run-time environment of arbitrary MPI implementations. Its design and implementation allows adaptation of this architecture to other implementations of MPI, such as MPICH (MPI Chameleon) [6] and OpenMPI [15].

3.1 Group Communication Framework and Fault Detector

Figure 3 depicts the framework for group communication implemented as a process of the lam daemon (lamd). The so-called membership daemons in the LAM universe communicate with each other and with the scheduler daemons through out-of-band communication channel provided by the LAM runtime environment.

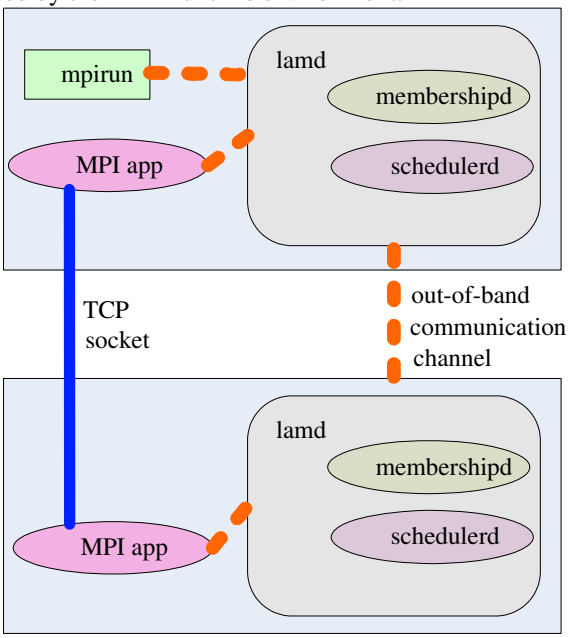

Figure 3. Group Membership and Scheduler

3.2 Internal Schedule Mechanism in LAM/MPI

As main part of the schedule mechanism, the scheduler daemon is also implemented as a process of lamd communicating through the out-of-band communication channel, just as the membership daemon does (Figure 3).

When the scheduler daemon receives a failure message from the membership daemon, it consults the database to retrieve information about the MPI job and the nodes in the LAM universe. Based on this information, the most lightly loaded node is chosen as a migration target. Alternate selection policies to determine a replacement node can be readily

plugged in. Next, a job pause command is issued to *mpirun* through the scheduler daemon at the node on which *mpirun* is launched.

Triggered by the scheduler, *mpirun* retrieves information from an application schema file created during the last checkpoint. This application schema specifies the arguments required to initiate a *cr_restart* on specified nodes. We implemented an extension to *mpirun* that modifies and enhances the application schema at run-time with the following information received from the scheduler:

- For the processes on the live nodes, replace *cr_restart* with *cr_pause* using the respective arguments to the command;
- For the processes to migrate, replace the node number of the failed node with the replacement node. Update naming references referring to nodes with regard to this file to reflect any migrations.

3.3 Job-pause

The pause implementation in LAM/MPI relies on the out-of-band communication channel provided by lamd. Its design is not constrained to the interfaces of LAM/MPI and may be easily retargeted to other MPI or C/R implementations. In our case, *crtcp* [25] and *cr* [24] are utilized, which represent the interface to the most commonly used C/R mechanism provided by LAM (while other C/R mechanisms may coexist).

When a node fails or a communication link lapses, the MPI-related processes on other live nodes will block/suspend waiting for the failure to be addressed, which realizes the passive facet of the job pause mechanism. On the active side, we restore the failed process image from a checkpoint file. This checkpoint information encompasses, among other data, pending MPI messages. Hence, in-flight data on the network does not unnecessarily have to be drained at pause time. Consequently, a process can be individually paused at arbitrary points during its execution.

Figure 4 shows the steps involved during the job pause in reference to BLCR. A detailed account of the individual events during checkpointing and restarting is given in the context of Figures 1 and 2 of [12] and is abbreviated here due to space constraints. Our focus is on the enhancements to BLCR (large dashed box).

In the figure, time flows from top to bottom, and the processes and threads involved in the pause are placed from right to left. Activities performed in the kernel are surrounded by dotted lines. The callback thread (right side) is spawned as the application registers a threaded callback and blocks in the kernel until a pause occurs. When *mpirun* invokes the *pause* command of BLCR, it provides the process id and the name of a checkpoint file to *pause* as an argument. In response, the *pause* mechanism issues an *ioctl* call, thereby resuming the callback thread that was previously blocked in the kernel. After the callback thread invokes the

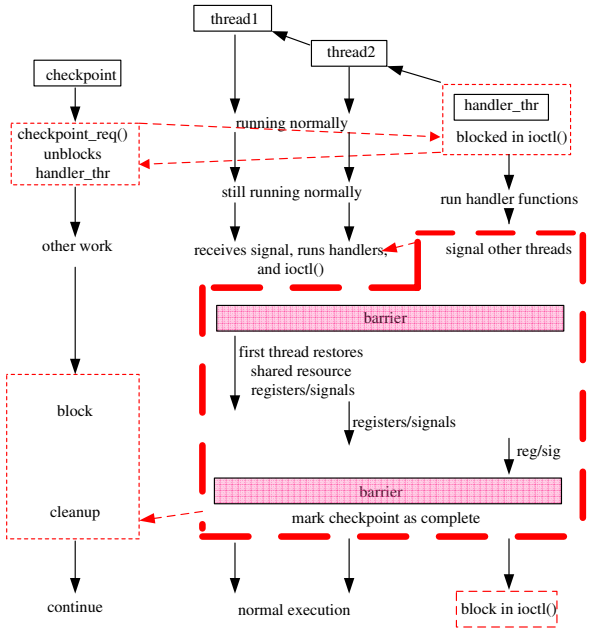

Figure 4. BLCR with Pause in Bold Frame

individual callback for each of the other threads, it reenters the kernel and sends a pause signal to each threads. These threads, in response, engage in executing the callback signal handler and then enter the kernel through another *ioctl* call.

Once in the kernel, the threads invoke the *thaw* command using VMADump to take turns reading their register and signal information from the checkpoint file. In our implementation, they no longer need to restore their process IDs and process relationships as we retain this information. After a final barrier, the process exits the kernel and enters user space, at which point the pause mechanism has completed.

Once the MPI processes have completed the pause and resume operation, they will wait for *mpirun* to supply information about the migrated process before establishing connections with the migrated process based on the received information. Once the connections have been established, the processes continue their normal execution.

3.4 Process Migration

Process migration after a failure is initiated on the replacement node when it receives a restart request from *mpirun*. Within the callback handler, the migrated process relays its addressing information to *mpirun*. This addressing information can later be identified as type *struct_gps* in the LAM universe. Next, *mpirun* broadcasts this information to all other processes and, conversely, sends information about all other processes (generated from the application schema) to the migrated process.

Once the processes receive the addressing information from *mpirun*, they reset the information in their local process list before establishing a TCP connection between each other. Notice that the connections among the paused pro-

cesses remain active, *i.e.*, only the connections between the migrated process and the paused processes need to be established.

Another enhancement with LAM/MPI for process migration is accomplished as follows. The LAM daemon establishes a named socket visible through the local file system (typically under the name `/tmp/lam-<username>@<hostname>/` or, alternatively, location-dependent on the TMPDIR environment variable or the LAM_MPI_SESSION_PREFIX variable). Since LAM initializes this location information at startup time and never refreshes it thereafter, we force a reset of the naming information at the replacement node resulting in an update of the directory reference inside the callback function of the migrated process.

The functionality to realize process migration discussed so far enhances the LAM/MPI runtime system. In addition, upon restarting a BLCR-checkpointed job on a different node, we must ensure that the operating system on all nodes supplies the exact same libraries across migration. Any library reference by an executable that is migrated has to be portable across migrations. BLCR does not save state of shared libraries (*e.g.*, initialization state of library-specific variables). As of late, some distributions of Linux are using "prelinking" to assign fixed addresses for shared libraries in a manner where these fixed addresses are randomized to counter security attacks. We had to deactivate the prelinking feature in our system to provide cross-node compatible library addresses suitable for process migration.

4 Experimental Framework

We conducted our performance evaluations on a local cluster that we control. This cluster has sixteen compute nodes running Fedora Core 5 Linux x86_64 (Linux kernel-2.6.16) connected by a Gigabit Ethernet switch. Each node in the cluster is equipped with four 1.76GHz processing cores (2-way SMP with dual-core AMD Opteron 265 processors) and 2 GB memory. A 750 GB RAID5 array provides shared file service through NFS over the Gigabit switch, which is configured to be shared with MPI traffic in these experiments. We extended the latest versions of LAM/MPI (lam-7.2b1r10202) and BLCR (blcr-0.4.pre3_snapshot_2006_09_26) with our job pause mechanism for this platform.

5 Experimental Results

We assessed the performance of our system in terms of the time to tolerate faults for MPI jobs using the NAS Parallel Benchmarks (NPB) [31]. NPB is a suite of programs widely used to evaluate the performance of parallel systems. The suite consists of five kernels (CG, EP, FT, MG, and IS) and three pseudo-applications (BT, LU, and SP).

In the experiments, the NPB suite was exposed to class C inputs running on 4, 8, and 16 nodes. IS was excluded from experiments due to its extremely short completion time ($\approx$10 seconds on 16 nodes), which did not reliably allow us to checkpoint and restart (with about the same overhead). All job pause/restart results were obtained from five samples with a confidence interval of ± 0.1s for short jobs and ± 3s for long jobs with a 99% confidence level.

Prior work already focused on assessing the cost of communication within the LAM/MPI and BLCR environments to checkpoint and restart MPI jobs [12, 10]. Our job pause and process migration mechanism decreases the communication by avoiding to re-establish the connections among the paused processes. Yet, the benefits of our approach lie in its applicability to proactive fault tolerance and incremental checkpointing. Our experiments are targeted at capturing the overhead of our approach, comparing it to prior approaches and analyzing the cause of its cost.

5.1 Checkpointing Overhead

Our system periodically takes snapshots of the MPI jobs at checkpoints yielding a transparent fault tolerance mechanism. Jobs can automatically recover by continuing from the most recent snapshot when a node fails. Figures 5, 6, and 7 depict the checkpoint overhead for 4, 8 and 16 nodes. As shown by these results, the overhead of job-pause C/R is uniformly small relative to the overall execution time of a job (benchmark), even for a larger number of nodes.

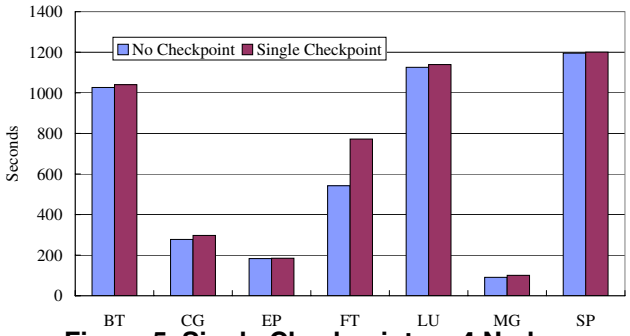

Figure 5. Single Checkpoint on 4 Nodes

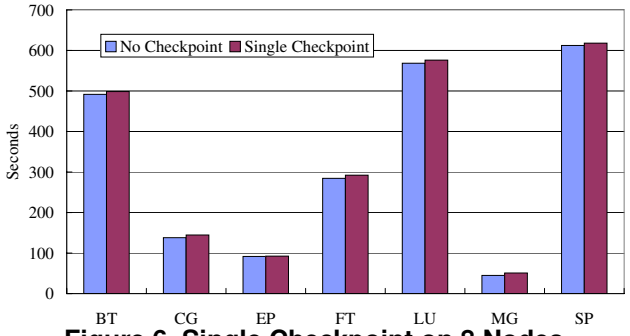

Figure 6. Single Checkpoint on 8 Nodes

Figure 8 depicts the measured overhead for single checkpointing relative to the base execution time of each benchmark (without checkpointing). For most benchmarks, the ratio is below 10%. Figure 9 depicts the corresponding time

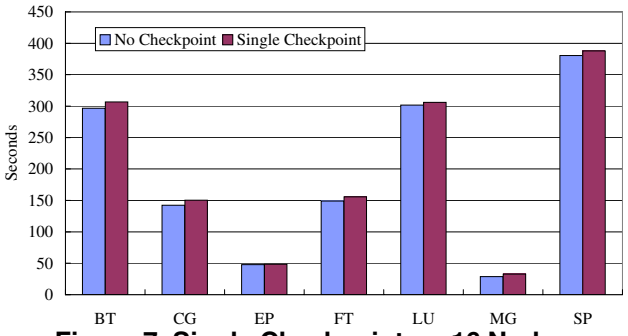

Figure 7. Single Checkpoint on 16 Nodes

for single checkpointing. This time is in the order of 1-12 seconds, except for MG and FT as discussed in the following.

MG has a larger checkpoint overhead (large checkpoint file), but the ratio is skewed due to a short overall execution time (see previous figures). In practice, with more realistic and longer checkpoint intervals, a checkpoint would not be necessitated within the application's execution. Instead, the application would have been restarted from scratch. For longer runs with larger inputs of MG, the fraction of checkpoint/migration overhead would have been much smaller.

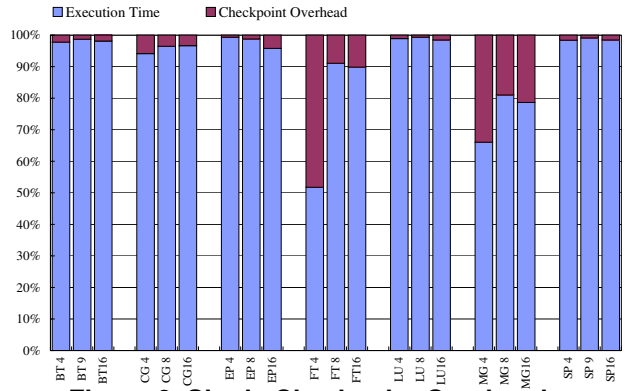

Figure 8. Single Checkpoint Overhead

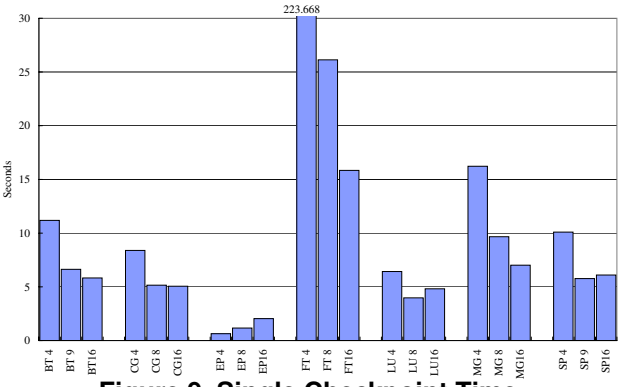

Figure 9. Single Checkpoint Time

As shown in prior work [12], checkpoint times increase linearly with the application's virtual memory (VM) size for processes that consume significantly less than half of the system's physical memory. However, when a process utilizes a VM size approaching or exceeding half the physical memory on the system, overheads increase up to an order of magnitude. This artifact explains the relatively high cost of FT for checkpointing (Figures 5, 8 and 9) and pause/restart (Figure 10) when it is run on just four nodes. For a larger number of nodes, the problem size is split such that smaller amounts of the VM are utilizes, which eliminates this problem. This property of checkpoint overhead relative to VM utilization is not inherent to our solution; rather, it is inherent to the checkpoint mechanism used in combination with the operating system.

Table 1 depicts the size of the checkpoint files for one process of each MPI application. The average file size is 135 MB on 16 nodes, 259 MB for 8 nodes, and 522.91 MB for 4 nodes. Writing many files of such size to shared storage synchronously may be feasible for high-bandwidth parallel file systems. In the absence of sufficient bandwidth for simultaneous writes, we provide a multi-stage solution where we first checkpoint to local storage. After local checkpointing, files will be asynchronously copied to shared storage, an activity governed by the scheduler. This copy operation can be staggered (again governed by the scheduler) between nodes. Upon failure, a spare node restores data from the shared file system while the remaining nodes roll back using the checkpoint file on local storage, which results in less network traffic.

Table 1. Size of Checkpoint Files $[MB]$

Node #	BT	CG	EP	FT	LU	MG	SP
4	406.90	250.88	1.33	1841.02	185.51	619.46	355.27
8	186.68	127.17	1.33	920.82	99.50	310.36	170.47
16	111.12	63.50	1.33	460.73	52.61	157.31	100.39

Overall, the experiments show that the checkpoint/restart overhead of the MPI job

1. is largely proportional to the size of the checkpoint file (Table 1 and Figure 9); and
2. is nearly the same at any time of the execution of the job.

The first observation indicates that the ratio of communication overhead to computation overhead for checkpoint/restart of the MPI job is relatively low. Since checkpoint files are, on average, large, the time spent on storing/restoring checkpoints to/from disk accounts for most of the measured overhead. This overhead can be further reduced. We are currently investigating the potential for savings through incremental checkpointing and proactive diskless live migration, which would reduce the checkpoint and pause overheads, respectively.

In our experiments, communication overhead of the applications was not observed to significantly contribute to the

In our experiments, we simply copy exactly one checkpoint file to shared storage, namely the file of the node whose image will be migrated.

overhead or interfere with checkpointing. This is, in part, due to our approach of restarting from the last checkpoint. Hence, our autonomic fault-tolerance solution should scale to larger clusters.

The second observation about checkpoint overheads above indicated that the size of the checkpoint file remains stable during job execution. The NPB codes do not allocate or free heap memory dynamically within timesteps of execution; instead, all allocation is done during initialization, which is typical for most parallel codes (except for adaptive codes [30]). Thus, we can assume the time spent on checkpoint/restart is constant. This assumption is critical to determine the optimal checkpoint frequency [32].

5.2 Overhead of Job Pause

As mentioned in the implementation section, the job pause mechanism comprises two parts: *cr_pause* within BLCR and the pause work at the LAM level. Experiments indicated that the overhead of *cr_pause* is almost the same as that of *cr_restart*. Though we reuse existing processes and avoid restoring the parent/child relationship as *cr_restart* does, we still need to restore the entire shared state (mmaps, files, etc.), which is generally large and accounts for the main overhead of *cr_pause/cr_restart*. The overall effect on our scheme is, however, not bounded by the roll-back on operational nodes. Instead, it is constrained by process migration, which uses *cr_restart* to restore the process from the checkpoint file on the replacement node. The pause mechanism at the LAM level actually saves the overhead to reconnect sockets between the processes. This is reflected in Figures 10, 11, and 12, which show that our approach is performing nearly at par with with the job restart overhead for 4, 8 and 16 nodes. Yet, while job restart requires extra overhead for rebooting the LAM subsystem of processes, our approach does not incur this cost as it reuses existing processes.

5.3 Membership/Scheduler Performance

The decentralized group membership protocol, adopted from our prior work [29] and integrated in LAM, has been shown to yield response times in the order of hundreds of microseconds and single-digit milliseconds for any reconfiguration (node failure) under MPI. This overhead is so small that it may be ignored when considering the restart overhead in the order of seconds / tens of seconds.

The overhead of our new scheduler is also small. The impact of scheduling is actually spread over the entire execution of the MPI job. Yet, when considering fault tolerance, only the overheads to (a) determine the replacement node and (b) trigger *mpirun* need to be accounted for. Any global information about the MPI job is maintained by *mpirun* at job initiation and termination. The overhead to determine the replacement node and to trigger *mpirun* is also in the order of hundreds of microseconds and, hence, does not significantly contribute to the overall overhead of C/R.

5.4 Job Migration Overhead

The overhead of process migration represents the bottleneck of our fault tolerance approach. The job-pause mechanism of other (non-failed) nodes results in lower overhead, and the overhead of the group membership protocol and our novel scheduler is insignificant, as explained above. Let us consider the potential of our approach for job migration in contrast to a full restart. Figures 10, 11, and 12 show that the performance of job pause is only 5.6% larger than a complete job restart (on average for 4, 8 and 16 nodes). Yet, job pause alleviates the need to reboot the LAM run-time environment, which accounts for 1.22, 2.85 and 6.08 seconds for 4, 8 and 16 nodes. Hence, pause effectively reduces the overall overhead relative to restart.

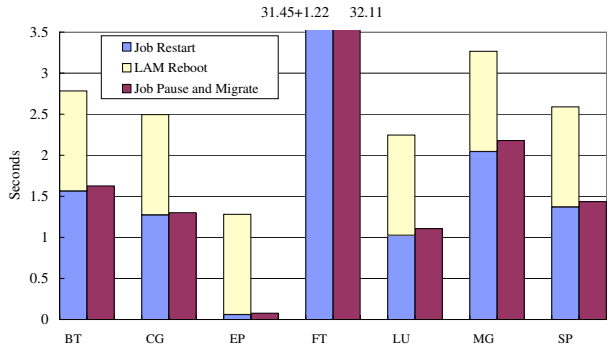

Figure 10. Pause and Migrate on 4 Nodes

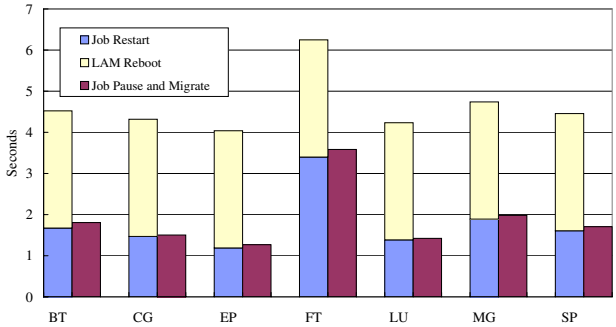

Figure 11. Pause and Migrate on 8 Nodes

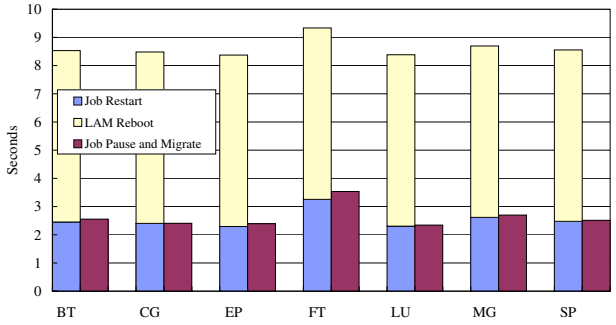

Figure 12. Pause and Migrate on 16 Nodes

Our approach has several operational advantages over a complete restart. First, job pause enables seamless and transparent continuation of execution across node failures. In contrast, a complete restart may be associated with a lengthy requeuing overhead in the job submission system.

Second, our approach is suitable for proactive fault tolerance with diskless migration. In such a scenario, healthy nodes would not need to roll back their computation. Instead, only the image of the unhealthy node is sent to a replacement node, which seamlessly picks up computation from there. The only effect on healthy nodes is that their socket connections have to be updated, which our scheme already supports. We are pursuing this approach, which with our novel job pause mechanism can now be realized.

The behavior of FT in Figures 10, 11, and 12, which differs from other codes, can be explained as follows. As previously mentioned, the checkpoint file of FT for 4 nodes exceeds half the physical memory. As a result, its pause and restart time goes up by an order of magnitude, as documented previously in the BLCR work [12].

Furthermore, consider the overhead of EP in Figures 9, 10, 11, 12. The checkpoint/pause&migrate/restart overhead of EP is becoming more dominant as we increase the number of nodes. This is caused by the small footprint of the checkpoint file (about 1 MB), which results in a relatively small overhead for storing/restoring the checkpoint file. Thus, the overhead of the migration combined with job pause mainly reflects the variance of the communication overhead inherent to the benchmark, which increases with the node count.

6 Related Work

A wide range of methods and systems to support fault tolerance (FT) have been developed in the past [23, 1, 5, 14, 17, 20, 7, 8, 11], mostly in the context of middleware where scalability was not the prime objective. Later approaches addressed this shortcoming and considered point-to-point as well as collective communication in the context of MPI [9, 2, 3]. Our work enhances LAM/MPI [24], which previously did not scale well at 1000 nodes or more, with a scalable group communication subsystem based on our prior work [29]. Besides scalability, our subsystem supports FT to sustain node failures.

A number of systems have been developed that combine FT with the message passing implementation of MPI, ranging from automatic methods (checkpoint-based or log-based) [27, 24, 6] to non-automated approaches [4, 13]. System checkpoint-based methods commonly rely on a combination of operating-system support to checkpoint a process image (*e.g., via* the BLCR Linux module [12]) combined with a coordinated checkpoint negotiation using collective communication among MPI tasks. User-level checkpointing, in contrast, relies on runtime library support and may require source preprocessing of an application and typically inflicts lower overhead, yet fails to capture critical system resources, such as file descriptors [18, 22]. Log-based methods rely on logging messages and possibly their temporal ordering, where the latter is required for asynchronous non-coordinated checkpointing. MPICH-V [6] implements three such protocols. It uses Condor's user-level checkpoint library [19]. Non-automatic approaches generally involve explicit invocation of checkpoint routines.

Our focus is on LAM/MPI+BLCR (coordinated system-level checkpointing) [26, 24]. LAM/MPI+BLCR requires a complete system restart where target node information, such as IP addresses, cannot be changed and checkpointing is not fully automated. This severely limits its applicability. Prior work extended this LAM/MPI+BLCR functionality to support migration of selected checkpoint images to new nodes [10]. Similar work extended the HA-OSCAR (High Availability Open Source Cluster Application Resources) distribution not only with compute-node failover (equivalent to migration with a complete start) but also to head-node failover (active-standby) in clusters [28]. This required modifications to LAM's hard-coded internal addressing information within the checkpointed file images followed by a complete job restart. Job submission frameworks, such as Torque, are oblivious to such changes if the new job is carefully constructed such as to resemble the originally submitted one. In contrast to this work, we take LAM/MPI+BLCR to yet another level with our novel job pause mechanism that supports migration without restart, *i.e.*, by retaining functioning process images and rolling back to the last checkpoint. Our approach is independent and transparent of any higher-level frameworks, such as job submission frameworks. More importantly, users do not lose their allocated time of a running job, *i.e.*, instead of requeuing the restarted job and waiting for its execution, execution commences from the last checkpoint without noticeable interruption.

7 Conclusion

This work contributes a fresh approach for transparent C/R through our novel job pause mechanism. The mechanism, implemented within LAM/MPI+BLCR, allows live nodes to remain active and roll back to the last checkpoint while failed nodes are dynamically replaced by spares before resuming from the last checkpoint. Enhancements to LAM/MPI include (1) support of scalable group communication with fluctuating number of nodes, (2) transparent coordinated checkpointing, (3) reuse of network connections upon failures for operational nodes, and (4) a BLCR enhancement for the job pause mechanism. We have conducted experiments with the NAS Parallel Benchmark suite in a 16-node dual-processor Opteron cluster. Results indicate that the performance of job pause is comparable to that of a complete job restart, albeit at full transparency and automation. A minimal overhead of 5.6% is only incurred in case migration takes place while the regular checkpoint overhead remains unchanged. Yet, our approach alleviates the need to reboot the LAM run-time environment, which accounts for considerable overhead resulting in net savings of our scheme in the experiments. Furthermore, job

pause reuses existing resources and continues to run within the scheduled job, which can avoid staging overhead and lengthy requeuing in submission queues associated with traditional job restarts. Our experiments also indicate that, after the initialization phase, checkpoints are constant in size for a given application, regardless of the timing of checkpoints. Our job pause approach further offers an additional potential for savings through incremental checkpointing and proactive diskless live migration, both of which are subject to future work.

References

[1] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, Nov. 1995.

[2] T. Angskun, G. Fagg, G. Bosilca, J. Pjesivac-Grbovic, and J. Dongarra. Scalable fault tolerant protocol for paralle runtime environments. In *Ero PVM/MPI*, 2006.

[3] T. Angskun, G. Fagg, G. Bosilca, J. Pjesivac-Grbovic, and J. Dongarra. Self-healing network for scalable fault tolerant runtime environments. In *Austrian-Hungarian Workshop on Distributed and Parallel Systems*, 2006.

[4] R. T. Aulwes, D. J. Daniel, N. N. Desai, R. L. Graham, L. D. Risinger, M. A. Taylor, T. S. Woodall, and M. W. Sukalski. Architecture of LA-MPI, a network-fault-tolerant MPI. In *Int'l Parallel and Distributed Processing Symposium*, 2004.

[5] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu. Coyote: a system for constructing fine-grain configurable communication services. *ACM Trans. Comput. Syst.*, 16(4):321–366, 1998.

[6] G. Bosilca, A. Boutellier, and F. Cappello. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing*, Nov. 2002.

[7] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of MPI programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2003.

[8] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Collective operations in an application-level fault tolerant MPI system. In *International Conference on Supercomputing*, June 2003.

[9] R. Butler, W. Gropp, and E. L. Lusk. A scalable process-management environment for parallel programs. In *Euro PVM/MPI*, pages 168–175, 2000.

[10] J. Cao, Y. Li, and M. Guo. Process migration for mpi applications based on coordinated checkpoint. In *Int'l Conference on Parallel and Distributed Systems*, pages 306–312, 2005.

[11] S. Chakravorty, C. Mendes, and L. Kale. Proactive fault tolerance in large systems. In *Workshop on High Performance Computing Reliability Issues*, 2005.

[12] J. Duell. The design and implementation of berkeley lab's linux checkpoint/restart. Tr, Lawrence Berkeley National Laboratory, 2000.

[13] G. E. Fagg and J. J. Dongarra. FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world. In *Euro PVM/MPI User's Group Meeting, Lecture Notes in Computer Science*, volume 1908, pages 346–353, 2000.

[14] R. Friedman and R. van Renesse. Strong and weak virtual synchrony in Horus. Technical Report TR95-1537, Cornell University, Computer Science Department, Aug. 24, 1995.

[15] J. Hursey, J. M. Squyres, and A. Lumsdaine. A checkpoint and restart service specification for open mpi. Technical report, Indiana University, Computer Science Department, 2006.

[16] IBM T.J. Watson. Personal communications. Ruud Haring, July 2005.

[17] I. Keidar. Group communication, June 12 2000.

[18] M. Litzkow. Remote unix - turning idle workstations into cycle servers. In *Usenix Summer Conference*, pages 381–384, 1987.

[19] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[20] D. Malki, D. Dolev, and R. Strong. A framework for partitionable membership service, Aug. 19 1995.

[21] I. Philp. Software failures and the road to a petaflop machine. In *Workshop on High Performance Computing Reliability Issues*. IEEE Computer Society, 2005.

[22] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

[23] M. L. Powell and B. P. Miller. Process migration in DEMOS/MP. In *Symposium on Operating Systems Principles*, pages 110–119, Oct. 1983.

[24] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.

[25] J. M. Squyres, B. Barrett, and A. Lumsdaine. Request progression interface (RPI) system services interface (SSI) modules for LAM/MPI. Technical Report TR579, Indiana University, Computer Science Department, 2003.

[26] J. M. Squyres and A. Lumsdaine. A component architecture for lam/mpi. In *European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387. Springer-Verlag, Sep/Oct 2003.

[27] G. Stellner. CoCheck: checkpointing and process migration for MPI. In IEEE, editor, *International Parallel Processing Symposium*, pages 526–531, 1996.

[28] A. Tikotekar, C. Leangsuksun, and S. L. Scott. On the survivability of standard mpi applications. In *Int'l Conference on Linux Clusters: The HPC Revolution*, May 2006.

[29] J. Varma, C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Scalable, fault-tolerant membership for mpi tasks on hpc systems. In *International Conference on Supercomputing*, pages 219–228, June 2006.

[30] A. Wissink, R. Hornung, S. Kohn, and S. Smith. Large scale parallel structured amr calculations using the samrai framework. In *Supercomputing*, Nov. 2001.

[31] F. Wong, R. Martin, R. Arpaci-Dusseau, and D. Culler. Architectural requirements and scalability of the NAS parallel benchmarks. In *Supercomputing*, 1999.

[32] J. W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, 1974.