ABSTRACT

SAINI, AJAY. Affinity-Aware Checkpoint Restart. (Under the direction of Dr. Frank Mueller.)

Current checkpointing techniques employed to overcome faults for HPC applications result in inferior application performance after restart from a checkpoint for a number of applications. This is due to a lack of page and core affinity awareness of the checkpoint/restart (C/R) mechanism, i.e., application tasks originally pinned to cores may be restarted on different cores, and in case of non-uniform memory architectures (NUMA), quite common nowadays, memory pages associated with tasks on a NUMA node may be associated with a different NUMA node after restart. This work contributes a novel design technique for C/R mechanisms to preserve task-to-core maps and NUMA node specific page affinities across restarts. Experimental results with BLCR, a C/R mechanism, enhanced with affinity awareness demonstrate significant performance benefits with negligible overheads compared to its affinity-agnostic counterpart.

Affinity-Aware Checkpoint Restart

by
Ajay Saini

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2014

APPROVED BY:

DEDICATION

To my parents.

BIOGRAPHY

Ajay Saini was born in Bahadurgarh, a small town in the state of Haryana in India. He did his schooling in Bahadurgarh and New Delhi and went to YMCA University of Science and Technology Faridabad, Haryana for his B.Tech in Computer Science. He joined ST Microelecronics Pvt. Ltd. as a Systems Engineer and worked there for three years. He came to NC State in Fall 2012 as a Master's student in the department of Computer Science. He has been working under Dr. Frank Mueller as a Research Assistant since August 2013.

ACKNOWLEDGEMENTS

This work would not have been possible without the collective effort of a lot of people. First and foremost, I would like to thank my advisor Dr. Frank Mueller for showing confidence in me and giving me the opportunity to work on this project. His guidance and feedback put me on the right track from time to time. I would like to thank Eric Roman and Paul H. Hargrove at Lawrence Berkley National Laboratory for their advice and support. Special thanks to Arash Rezaei, PhD student under Dr. Frank Mueller, for his continuous support and guidance when I hit an impasse. I am thankful to Dr. William Enck and Dr. Steffen Heber to serve on my committee. Lastly, I would like to thank my labmates in the System Research Lab and my family for their support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 High Performance Computing (HPC)

HPC systems, also popularly referred to as Supercomputers, generally capitalize on aggregating computing power in a way that delivers much higher performance than one could get out of a typical single desktop computer or workstation in order to solve large problems in science, engineering, or business [1]. They are used for a wide range of computationally intensive tasks in various fields, including quantum mechanics, weather forecasting, climate research, oil and gas exploration, molecular modeling and physical simulations. HPC systems have been shifting from expensive massively parallel architectures to clusters of commodity PCs to take advantage of cost and performance benefits. The TOP500 [4] list provides rankings of the fastest supercomputers according to their LINPACK benchmark results twice a year.

The parallel architectures of supercomputers may require the use of special programming techniques to exploit their full potential. Message passing is one of the widely used programming paradigm for HPC systems. Application work is divided among nodes/systems and they communicate through messages to solve large problems. The Messsage passing Interface (MPI) is a standardized and portable message-passing API (Application Programming Interface) widely used in research and industry. The standard defines the syntax and semantics of core of library routines for writing portable message-passing programs in Fortran or C/C++. Several mature and efficient implementations of MPI exists, including OpenMPI, MPICH and MVAPICH. More and more applications utilize MPI to exploit the computing power of HPC systems.

1.2 Fault Tolerance

HPC systems have large computing power. But at large scale, systems become unreliable as their component count (10,000s or even 100,000s of processing cores) increases the overall fault rates. HPC systems may experience both software faults and hardware faults. The mean time between failures (MTBF) / interrupts (MTBI) is in the range of 6.5-40 hours depending on the maturity / age of HPC installations [28]. Faults may result in a lost computation or, if handled via fault tolerance, in an increase in execution time and cost of running the applications. Fault tolerance is an active area of research which pertains to improving reliability of HPC systems.

1.3 Checkpoint/Restart and BLCR

Checkpoint/Restart is one of the most widely used mechanism for fault tolerance for HPC systems. Using this mechanism, the state of the application is saved at regular intervals on stable storage. When a fault is encountered, then instead of restarting the application from the beginning, it can be started from the last checkpoint. This saves the cost and time required to re-execute the entire application when faults occur.

BLCR [11] is one such checkpoint/restart mechanism. It is implemented as a Linux kernel module combined with a user-level shared library. It can checkpoint a multi-threaded application and its support has also been extended to MPI implementations like OpenMPI, enabling it to checkpoint multi-process applications. It provides command line tools and function calls to checkpoint/restart an application. Using simple commands, e.g., *cr_checkpoint* ⟨*processid*⟩, an application can be checkpointed and its state is written to a file on disk. In case of a fault, the application can be restarted from the checkpoint file using the *cr_restart* ⟨*checkpoint_file*⟩ command.

1.4 OpenMP

OpenMP [3] is an API jointly defined by a group of major computer hardware and software vendors with the goal of providing a standard among a variety of shared memory architectures and platforms. It is comprised of three primary API components: Compiler directives, runtime library routines and environment variables. The API supports C/C++ and Fortran on a wide variety of architectures. For example, in the *C* language, the gcc compiler flag *-fopenmp* and header file *omp.h* enables support for OpenMP.

OpenMP programs exploit parallelism exclusively through the use of threads. The programming paradigm uses the fork-join model of parallel execution (see Figure 1.1). All OpenMP programs begin as a single process, the master thread. The master thread executes sequentially until the first parallel region construct is encountered. The master thread then creates a team of parallel threads. The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the threads in a team. When the threads of a team complete the statements in the parallel region construct, they synchronize and terminate.

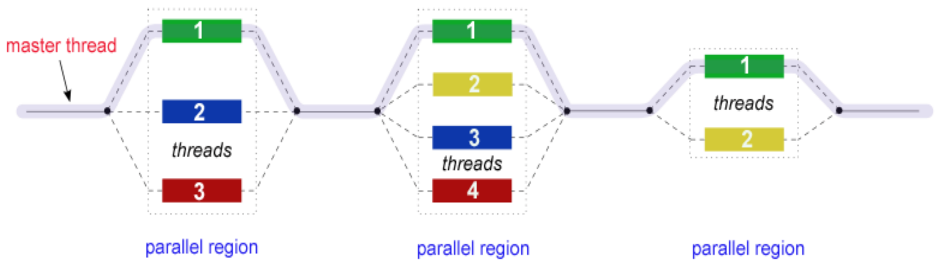
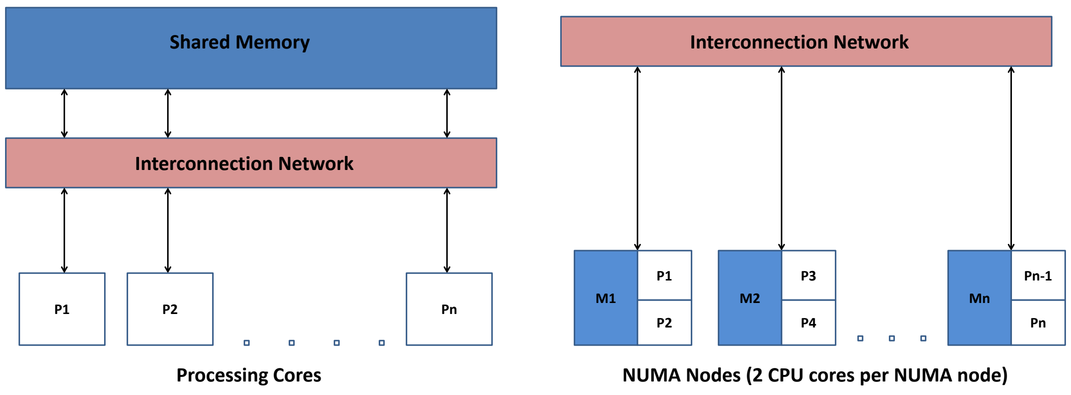


Figure 1.1: OpenMP: fork-join model of parallel execution [3]

Only the master thread continues to execute. The number of parallel regions and the threads that comprise them are configurable through compiler directives and/or environment variables. OpenMP is commonly employed to exploit parallelism in HPC applications.

1.5 NUMA Architectures

Modern CPUs operate at higher execution rates than main memory can be accessed. With the increase in the number of processing cores on a single system, uniform memory access architecture (Figure 1.2(a)) can starve a set of processors, notably because only one processor can access the computer's memory at a time. NUMA [2] (Figure 1.2(b)) attempts to address this shortcoming by attaching separate memory per processor (or group of cores), avoiding the performance hit when several processors attempt to address the same memory. NUMA is a computer memory design used in multiprocessing, where the memory access time depends on the distance to memory location relative to the requesting processor. A NUMA system classifies memory into NUMA nodes. A processor can access its own local NUMA node memory faster than non-local memory (memory assigned to a remote NUMA node). NUMA designs typically provide performance gains over other multiprocessing architectures, especially if a software application is designed to take advantage of such a memory hierarchy.



(a) Uniform Memory Access (b) Non-Uniform Memory Access

Figure 1.2: Memory Access Architectures

Chapter 2

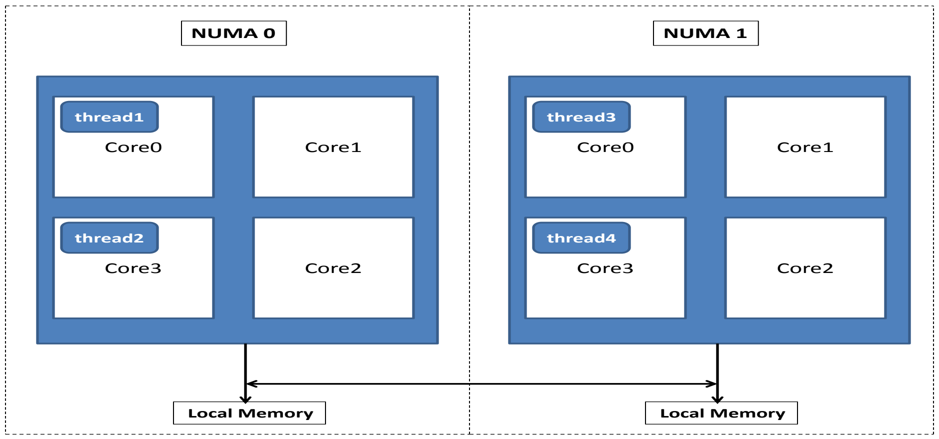
Motivation, Hypothesis and Contribution

2.1 Motivation

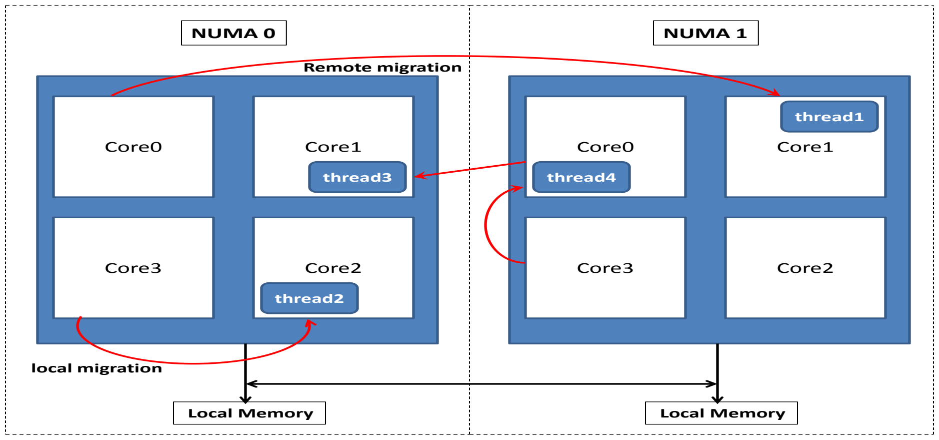
With the recent rise in the number of processing cores on HPC systems (10,000s or even 100,000s of processing cores), faults are also becoming common. The mean time between failures (MTBF) / interrupts (MTBI) is in the range of 6.5-40 hours depending on the maturity / age of the installation [28]. Several approaches have been studied to enable fault tolerance in an HPC environment. One of the widely used methods is Checkpoint/Restart (C/R). It involves saving the context of a job/application at regular intervals and restarting the application from a saved context if a failure occurs. Such an approach saves significant time because we do not have to start the job from scratch. A large number of checkpoint-restart utilities have been developed, each with its own advantages [22].

Another notable development attributed to the increase in the number of cores is an accelerated shift towards distributed non-uniform memory access (NUMA) architectures. Such architectures consist of collections of computing cores with fast local memory, communicating with each other via a slower inter-chip communication medium. Access by a core to the local memory, and in particular to a shared local cache, can be several times faster than access to the remote memory or cache lines resident on another chip.

Several applications developed for such HPC environments take advantage of this non-uniform memory arrangement. Applications, at times, are started with its threads pinned to particular cores. This preserves both thread-to-core affinity and data-to-NUMA node (page-to-NUMA node) affinity. Judicious bindings can improve performance by reducing resource contention (by spreading processes apart from one another), reducing migration overheads and NUMA remote memory access penalties (by reducing excessive process movement), or improving inter-process communications (by placing processes close to one another). Figure 2.1(a) shows an application with 4 threads pinned to particular cores. But when the same application is executed without pinning (Figure 2.1(b)), its threads can migrate both locally (within a NUMA node) or remotely (to another NUMA node). Such migrations might result in overheads as a thread might be moved away from ”hot” caches or local NUMA memory.



(a) Application threads pinned to CPU cores



(b) Application threads not pinned resulting in migrations

Figure 2.1: Effects of pinning and no-pinning

There are various real world scenarios where pinning is beneficial, especially for applications which are sensitive to such placement. For example, Dice et. al [10] talk about the benefits of NUMA aware locks. Even Operating System (OS) process schedulers have inbuilt intelligence to reduce migrations, and their memory allocators are NUMA aware such that data is allocated on a local NUMA node where a thread is running.

Considering these two locality aspects - affinity awareness and checkpoint restart, performance suffers when an affinity sensitive application is checkpointed, and later on restarted using existing C/R techniques. Existing C/R techniques do not take affinity information into account. Even if we start an application with its thread pinned as in Figure 2.1(a), when such an application is restarted from a checkpoint, we might end up with an application run as in Figure 2.1(b) as no pinning is enforced. This is the problem we target in this work: How can we ensure that affinity information is preserved across restarts?

2.2 Hypothesis

We hypothesize that an application can regain its original runtime performance after restart from a checkpoint if its affinity information, i.e., process-to-core affinity and memory page-to-NUMA node affinity, is restored.

2.3 Contributions

When we restart an application from a checkpoint, we want that application to exhibit the same affinity behavior it had before the checkpoint. In this work, we present a simple and novel approach to save and restore affinity information. We have implemented our design in BLCR, enhancing it to affinity-aware BLCR. BLCR [11] is a hybrid checkpoint restart mechanism for Linux and is implemented as a kernel module with a user level library. With our enhancements and through configurable options, applications can be checkpointed and restarted with affinity awareness, both thread-to-core and page-to-NUMA node affinity. Applications that are sensitive to CPU core pinning and NUMA memory placement experience significant benefits when using the affinity-aware BLCR. We have evaluated the benefits of our enhancements on the NAS Benchmark suite and see performance improvement ranging from 45% to 70% in application execution time after restart compared to using the original BLCR. To the best of our knowledge, we are first to implement such affinity awareness in a checkpoint restart mechanism.

Chapter 3

Design

In this section, we present a high-level overview of the BLCR design before focusing on our enhancements for affinity awareness. For a detailed description of the BLCR design, see [11].

BLCR is implemented as a Linux kernel module combined with a user level shared library. An application can be checkpointed (i.e., its current state is written to a file) and restarted from a checkpoint file. Figure 3.1 and Figure 3.2 show the checkpoint and restart flow along with a table of actions taken at each step. The table shows actions common to both implementations (the original BLCR and the affinity-aware BLCR) as well as actions taken individually. Time flows from top to bottom in each diagram. Activities performed in the checkpoint restart flow are represented by numbers and described in the right halves of the figures.

In the following, we first discuss the checkpoint and restart flow of the original BLCR. Then we describe our enhancements and present the checkpoint and restart flow of our modified BLCR. We use a running example of an application with three threads in our description.

3.1 Original BLCR Checkpoint-Restart Flow

3.1.1 Checkpoint flow

When a checkpoint request is triggered, it results in the following sequence of actions (see Figure 3.1):

Step1: After the initialization phase, one of the application threads is selected as a group leader and the other threads wait for a wake up signal from the group leader.

Step2: The leader thread records parent/child relationships and then reaches a barrier to wake up other threads. All threads then return from this barrier to reach another barrier.

Step3: While the other threads wait, the leader thread records its process id, register contents and signals. It then saves shared items, including dirty pages, virtual memory maps, mmaped files and protection flags in the checkpoint file. On reaching another barrier, it wakes up the other threads and waits for them to complete.

Step4 and Step5: All threads save their private data including process id, register contents and signals.

Step6: After all threads have reached the final barrier, they return from kernel space and the application continues.

3.1.2 Restart flow

Restarting from a checkpoint is largely the inverse of the checkpoint process. A restart request results in the following actions (see Figure 3.2):

Step1: Once the initialization phase is completed, the restart process performs an ioctl() call, which causes the process to be forked. The parent process returns to user space and waits for the restart to complete. The child is cloned as many times as there were threads in the original application that is being restarted. One thread is selected as group leader, and the other threads wait for a wake up signal from the leader thread.

Step2: The leader thread loads its register contents and signals. It unmaps the existing virtual memory areas and remaps the virtual memory areas based on the information stored in the checkpoint file. It loads the shared items including dirty pages and uses kernel support (*sys_mprotect* in Linux kernel) to restore protection flags. The leader thread then reaches a barrier to wake up other threads and waits for them to complete.

Step3 and Step4: All other threads reloads their private data including registers and signals.

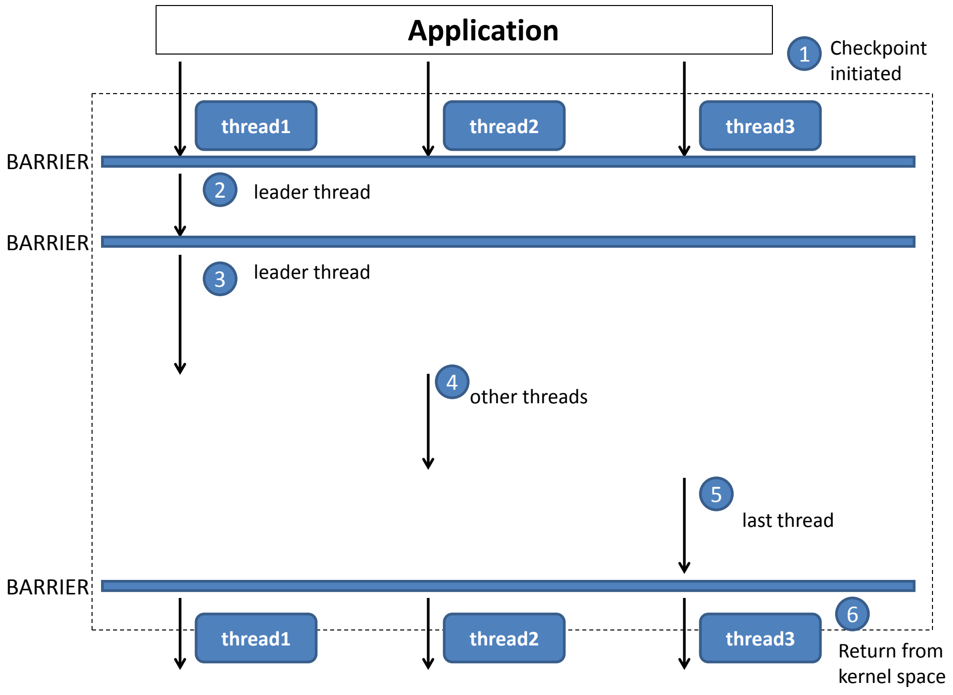
Step5: After all threads reach a barrier, the leader thread locks the kernel process table and restores the parent child relationship while the other threads wait. It then reaches a barrier, where it wakes up the other threads and waits for them to complete.

Step6: After all threads have reached the final barrier, they return from kernel space, i.e., the application is restarted and resumes normal execution.

3.2 Saving and Restoring Affinity Information

The checkpoint and restart flow described in the previous section does not consider affinity information, thread-to-core and page-to-NUMA node while checkpointing and restarting an application. To save and restore this affinity information, we considered various design approaches.

For thread to core affinity, we prototyped (1) a brute force approach, wherein we would extract and save the cpumask for each thread during checkpointing. During restart, we would

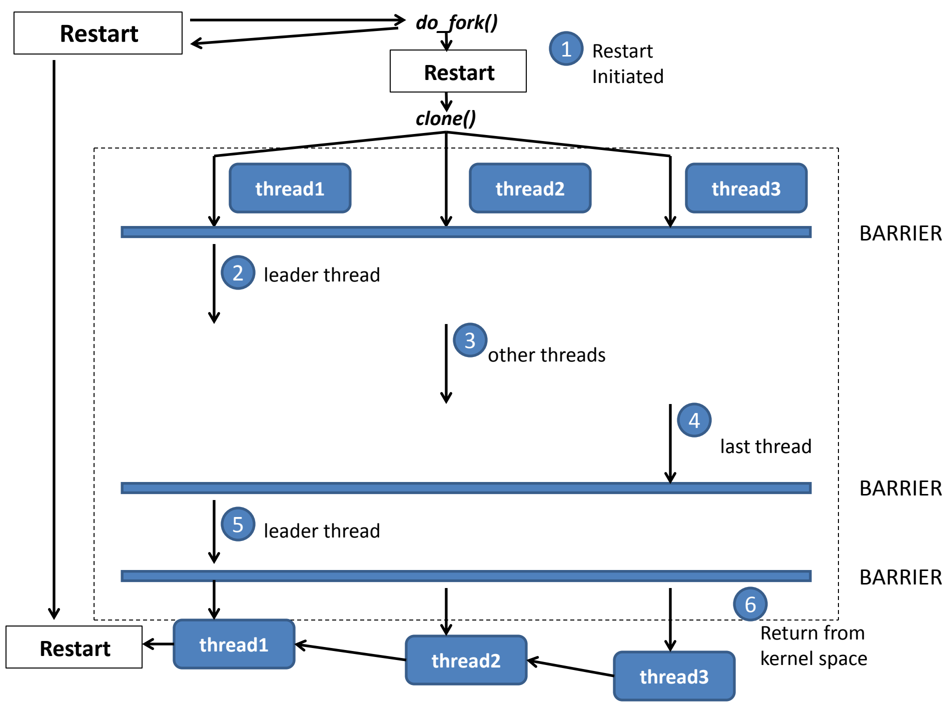


(a) Checkpoint flow diagram

Steps	Actions common to both implementations of checkpoint	Actions performed only by original BLCR checkpoint	Actions performed only by affinity-aware BLCR checkpoint
1	Checkpoint initiated		
2	leader thread records parent/child relationship		
3	leader thread records pid, registers, signals, shared items - mmaps, files, protection flags	leader thread saves all dirty pages	leader thread saves cpumask, dirty pages on local NUMA node only
4	other threads record pid, registers, signals.		other threads record cpumask, VM maps, protection flags, dirty pages on local NUMA node
5	last thread records pid, registers, signals.		last thread saves cpumask VM maps, protection flags, dirty pages on local + orphan NUMA node
6	Return from kernel space		

(b) Table of actions taken during checkpoint

Figure 3.1: Checkpoint flow



(a) Restart flow diagram

Steps	Actions common to both implementations of restart	Actions performed only by original BLCR restart	Actions performed only by affinity-aware BLCR restart
1	Restart initiated		
2	leader thread loads pid, registers, signals, unmaps VMA's and loads saved VMA's, files.	leader thread loads all the pages as it saved all of them, restores protection flags.	leader thread restores its cpumask, loads local NUMA saved pages
3	other threads load pid, registers, signals		other threads restore their cpumask and load local NUMA saved pages
4	last thread loads pid, registers, signals		last thread restores its cpumask and loads local + orphan NUMA saved pages, restores protection flags.
5	leader thread restores parent/child relationship		
6	Return from kernel space		

(b) Table of actions taken during restart

Figure 3.2: Restart flow

directly overwrite the cpumask of each thread with the saved one. This provisionally works on Linux, but may constrain portability as the state of a thread is modified without the kernel's knowledge. (2) We also tried to provide affinity information at the time of calling the clone function. But the Linux clone API lacks such a flag. We decided to implement a modification of the brute force approach. Instead of directly overwriting the cpumask, we would use kernel support to change the cpumask, making the method portable and kernel aware.

For page to NUMA node affinity, we prototyped (1) an approach to save the NUMA nodeid of pages while checkpointing. During restart, we use kernel support to migrate pages to appropriate NUMA nodes. On Linux, this method requires calls to unexported kernel functions from a kernel module and access to userspace buffer, both of which are not easily supported. Besides, we realized this approach would have page migration overheads. (2) Another approach was to divide the work of saving/loading pages among threads. This method was based on the fact that most of the operating systems, nowadays, are NUMA aware, i.e., their memory allocator allocates pages on the local NUMA node where the thread is running using the first touch policy. This approach does not inflict migration overheads. This is the approach we chose.

3.3 Affinity Aware BLCR Checkpoint-Restart Flow

With the above design schemes, we modified steps 3, 4 and 5 of BLCR's checkpoint flow. Correspondingly, to use this new stored information, we modified steps 2, 3 and 4 of the restart flow.

3.3.1 Checkpoint flow

A checkpoint request results in the following actions (see Figure 3.1):

Step1 and Step2: The same as earlier.

Step3: The leader thread, saves its process id, registers and signals. It also saves its cpumask. It then saves shared items, including virtual memory maps, mmaped files and protection flags, but only those dirty pages that belong to its local NUMA node. It then reaches a barrier where it wakes up other threads and waits for them to complete.

Step4: The other threads save their private data, including process id, registers and signals. Now, each thread also saves its cpumask and those dirty pages that belong to its local NUMA node, if these pages have not already been saved by another thread belonging to the same NUMA node. To maintain the original design flow and to account for the actions to be taken during restart, virtual memory maps and protection flags are also saved without incurring significant impact on the checkpoint file size.

Step5: The last thread, in addition to saving the same items as the other threads, also saves those pages that belongs to a NUMA node on which none of the threads are running (which we refer to as the orphan NUMA node).

Step6: The same as earlier.

3.3.2 Restart flow

A restart request results in the following actions (see Figure 3.2):

Step1: The same as earlier.

Step2: The leader thread loads the register contents, signals and cpumask. If cpumask is not equal to its current cpumask, it resets it using kernel support. It then unmaps the existing virtual memory areas and remaps the virtual memory areas using the stored information in the checkpoint file. It loads its saved pages. As the thread is running as per the original cpumask and, due to NUMA awareness of the memory allocator, pages are allocated on the same NUMA nodes that they were on before the checkpoint. The saved protection flags are not restored here. The leader thread then reaches a barrier where it wakes up other threads and waits for them to complete.

Step3: The other threads load their private data, including registers, signals and cpumask. Kernel support is needed to reset the cpumask. Then, the saved pages are loaded, and the pages are allocated on the correct NUMA nodes.

Step4: The last thread additionally brings the pages belonging to orphan NUMA nodes to its local NUMA memory and restores the protection flags, saved during checkpoint, for each of the virtual memory maps.

Step5 and Step6: The same as earlier.

Chapter 4

Implementation

We implemented our enhancements in BLCR on Linux. We took the following issues into consideration during our implementation. *First*, no changes should be made to the Linux kernel code. *Second*, implemented features of BLCR like synchronization, should be reused to keep the changes to minimum. *Third*, new updates should be flexible, i.e., they can be turned on/off via command line parameters and/or environment variables.

4.1 Saving and Restoring Thread to CPU Core Affinity

To save and restore thread-to-core affinity, we need kernel support to access this affinity information during checkpoint and reset it during restart. In the Linux kernel, CPU affinity is saved in the cpumask of a Task Control Block (TCB) of a thread/process. While checkpointing, we save this mask in the checkpoint file. During restart, we read this mask and then utilize the Linux kernel call

```
int set_cpus_allowed(struct task_struct *p, cpumask_t new_mask)
```

to reset the cpumask if the current mask is not equal to the saved one. This places the thread as per the new mask on the original CPU core.

4.2 Saving and Restoring Page to NUMA node Affinity

To save and restore page-to-NUMA node affinity, we rely on the first touch memory policy of Linux memory allocator, which allocates pages on the local NUMA node where the thread is running. In order to implement it, we need two pieces of information: (1) which NUMA node does a thread belong to and (2) which NUMA node does a page belong to. We utilize the kernel call

```
int numa_node_id(void)
```

to inquire which NUMA node the current thread is running on. To obtain the page-to-NUMA node mapping, we need page table information. BLCR already implements a page table walk function to extract page information given a virtual address. We utilize it to determine the page address and then use the kernel call

```
int page_to_nid(const struct page *page)
```

to determine the NUMA node of a page. Based on these two pieces of information, each thread only stores pages that belong to its local NUMA node. To avoid duplicate savings of pages, we use a flag array to keep track of NUMA nodes for which (potentially shared) pages have already been saved. Each thread consults this array before saving a page. The last thread additionally saves the leftover pages allocated on a NUMA node on which none of the threads are running (orphaned NUMA node). During restart, each thread starts loading pages after it has been rescheduled as per saved CPU affinity. Thus, each thread loads pages (utilizing the first touch memory policy) on the correct NUMA node maintaining the page-to-NUMA node affinity.

We have added support for environment variables and command line parameters to turn on/off these features. We have also added an additional environment variable to optionally reset the CPU affinity of the threads during restart. This allows threads to optionally be moved to a different set of CPU cores after restart.

Chapter 5

Results

5.1 Experimental Framework

Experiments were conducted on a node in a local cluster comprised of 108 compute nodes with 1728 cores. All machines are 2-way SMPs with AMD Opteron processors, eight 2GHZ cores per socket (16 cores per node) and four NUMA nodes (4 cores forming one NUMA node). Linux x86_64 version 2.6.32.27 is installed on each of the machines. The memory hierarchy consists of three levels of cache, L1 (64KB), L2 (512KB), L3 (5MB), and a 32GB RAM.

We used the OpenMP version of the NPB (NAS Parallel Benchmarks) suite [7], [14] (version 3.3) for our experiments. The NPB features a set of programs, BT, SP, LU, IS, FT, MG, CG, EP, DC and UA, designed to help evaluate the performance of parallel supercomputers. The benchmarks are derived from computational fluid dynamics (CFD) applications and originally consisted of five kernels (IS, FT, MG, CG, EP) and three pseudo-applications (BT, SP, LU). These five kernels mimic the computational core of five numerical methods used by CFD applications. The simulated CFD applications reproduce much of the data movement and computation found in full CFD codes. The benchmark suite has been extended to include two new benchmarks (UA, DC) for unstructured adaptive mesh, parallel I/O, multi-zone applications, and computational grids. We conducted experiments with BT, SP, LU, IS, FT, MG, CG, and UA. Others (SP, DC) did not have enough iterations to perform checkpoints/restarts.

We also present an analysis of affinity-aware BLCR for benchmarks/applications that are not sensitive to thread-to-core or page-to-NUMA node mappings. We observed LULESH benchmark to be one such example. LULESH [15] is a highly simplified kernel of an application, hard-coded to only solve a simple Sedov blast problem. LULESH approximates the hydrodynamics equations that describe the motion of materials relative to each other when subjected to forces.

Table 5.1: This table shows total iterations, and the iteration at which a checkpoint was initiated in our experiments

NAS Benchmarks	**Total iterations**	**Iteration no. at which a checkpoint was initiated**
BT	200	10
SP	400	10
FT	20	2
MG	20	2
LU	250	10
CG	75	10
UA	200	10
IS	10	2

5.2 Experiments

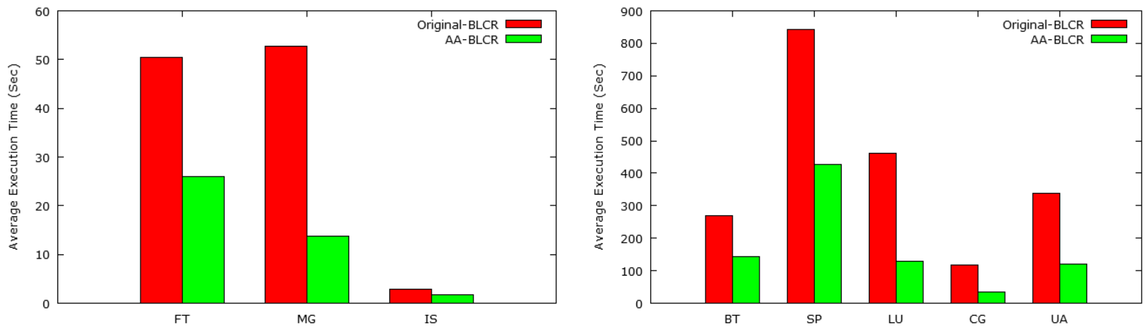
Experiments were conducted to assess (1) application execution time after restart from a checkpoint file, our major target area, (2) checkpoint file size overhead, (3) checkpoint time overhead, and (4) restart time overhead.

The input size for NAS parallel benchmarks can be configured as per different classes. We used CLASS C data inputs for our experiments as they had longer execution times and resulted in larger checkpoint files. While collecting data, we tried to remove background noise by starting with a fresh node (usually restarting that node) and fixing the CPU frequency for each of the cores to 2GHz before conducting our experiments.

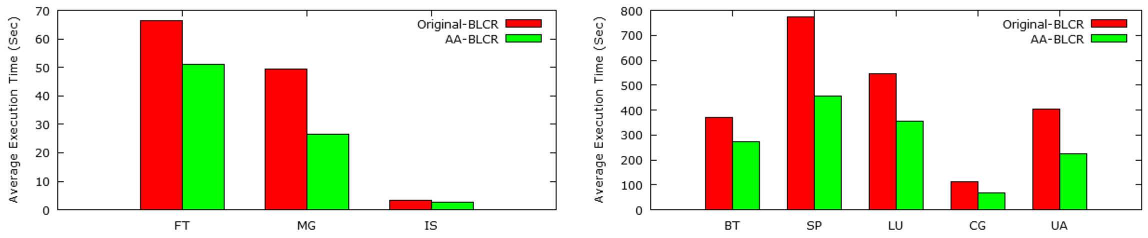
We instrumented the NAS benchmarks to initiate a checkpoint at a particular iteration count after initialization as shown in Table 5.1. The first column in Table 5.1 lists the NAS benchmarks, the second column shows the total iterations in the complete run of that benchmark and the third column shows the iteration number at which the checkpoint was initiated. In selecting an iteration to checkpoint at, we tried to ensure a sufficiently long execution time after restart. But as we found out in experiments, restarting from checkpoint taken, irrespective of the remaining work, resulted in similar performance.

During experiments with each benchmark, we took five different checkpoints at the same iteration count. Each checkpoint was restarted two times. This was done both for the original BLCR and the affinity-aware BLCR. The graphs depict average values of these runs. Percentage change between results of the original BLCR and the affinity-aware BLCR was calculated using the following formula:

$$\%Change = \frac{(OriginalBLCR\ Time\ -\ AffinityAwareBLCR\ Time) * 100}{OriginalBLCR\ Time} \tag{5.1}$$



(a) 16 Threads

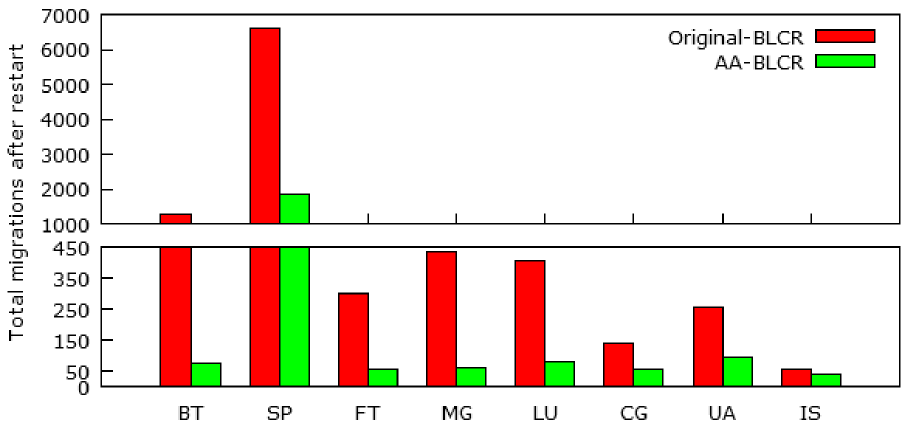


(b) 8 Threads

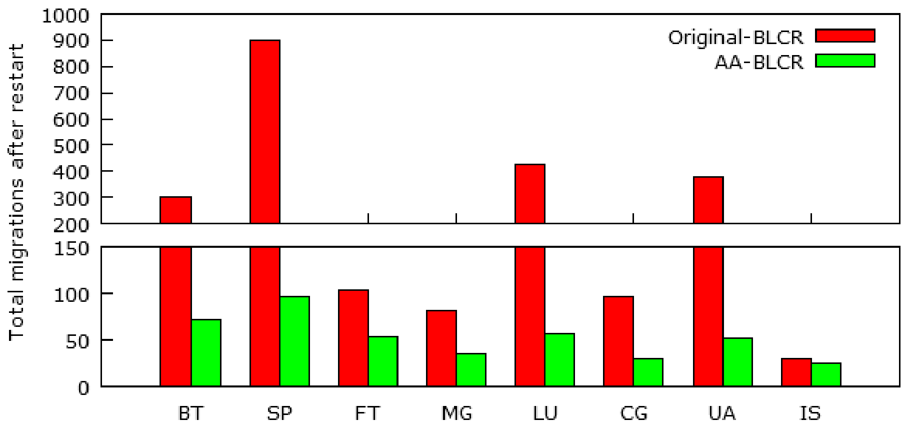
Figure 5.1: NAS Benchmark (CLASS C) execution time after restart (excluding restart time)

5.3 Performance

Figure 5.1 depicts the execution time of the NAS benchmarks after restart from a checkpoint file using the original BLCR and the affinity-aware BLCR (AA-BLCR). Figure 5.1(a) depicts results when each of the NAS benchmarks are configured to run with 16 threads. Figure 5.1(b) depicts results for 8 threads. The x-axis depicts each of the NAS benchmarks and the y-axis depicts the average execution time after restart in seconds. This execution time excludes the time taken by BLCR to restart the application. Due to the large range in the data values, the graph is split into two parts, with benchmarks FT, MG, IS depicted in the left half and BT, SP, LU, CG, UA depicted in the right half of Figure 5.1(a) and Figure 5.1(b). Table 5.2 depicts percentage change between the original BLCR and the affinity-aware BLCR for different parameters when using 16 threads. The first column lists the NAS benchmarks and the second

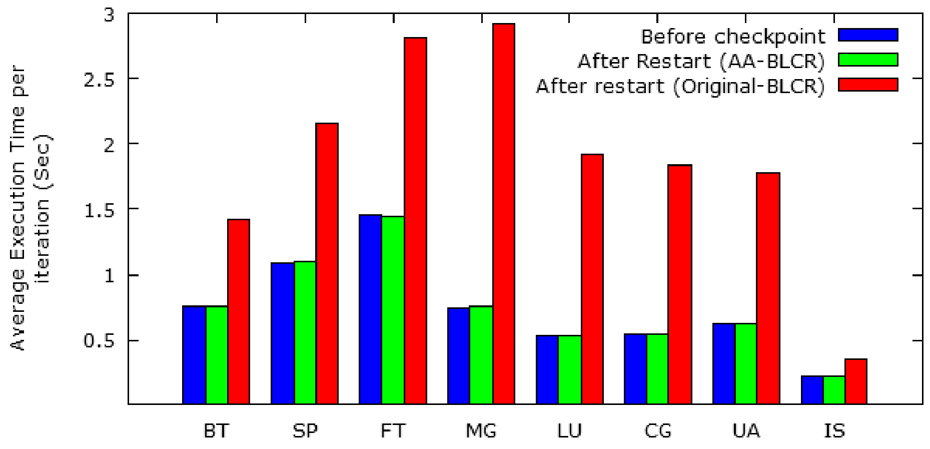


(a) 16 Threads

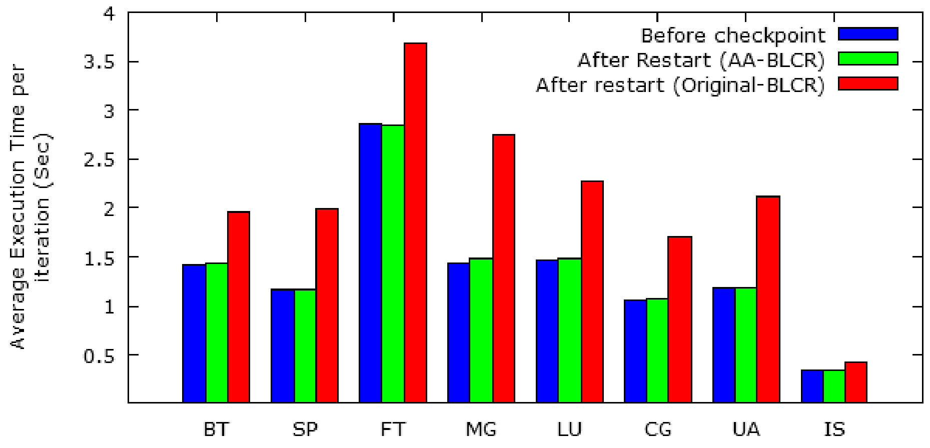


(b) 8 Threads

Figure 5.2: NAS Benchmark (CLASS C) total CPU migrations after restart



(a) 16 Threads



(b) 8 Threads

Figure 5.3: NAS Benchmark (CLASS C) average execution time of each iteration before checkpoint, after checkpoint using AA-BLCR and after checkpoint using Original-BLCR

column shows the percentage change in the application execution time after restart. Table 5.3 depicts these measurements for 8 threads. We observe significant improvements when using the affinity-aware BLCR for both thread configurations. The application execution time, after restart, improves between 37% and 73% for 16 threads (Table 5.2 second column) and between 18% and 46% for 8 threads (Table 5.3 second column).

In case of the original BLCR, affinity information is not saved and only the leader thread restores the memory information. This causes all the data to be allocated locally to a single NUMA node (the node the leader thread is running on), unless the leader thread scatters data over different NUMA nodes when occasionally migrated by the OS scheduler. When the application is restarted, threads may be scheduled to run on any core but when they try to access data, they suffer from NUMA remote memory access delays in addition to migration overheads, which causes the observed performance degradation. In case of the affinity-aware BLCR, affinity information is restored and no NUMA access delays and migrations are incurred. These observations are reflected in Figure 5.2 and Figure 5.3. Figure 5.2 depicts the total CPU migrations for the application after restart for the original BLCR and the affinity-aware BLCR on the y-axis for each NAS benchmarks on the x-axis. The *migrations* event of the *perf stat* command was used to obtain CPU migration numbers. Figure 5.2(a) depicts results for 16 threads and Figure 5.2(b) for 8 threads. Due to the large variations in the data values, the y-axis is divided into two intervals as shown in the Figure 5.2(a) and Figure 5.2(b). We observe a large reduction in CPU migrations for the affinity-aware BLCR as thread-to-core maps are restored. Table 5.2 (third column) depicts the percentage change in CPU migrations for 16 threads and Table 5.3 (third column) for 8 threads. There are still some migrations that are attributed to migrations before thread-to-core maps are restored. Figure 5.3 depicts the average execution time per iteration before initiating a checkpoint and after the restart from the checkpoint for the original BLCR and the affinity-aware BLCR on the y-axis for each NAS benchmarks on the x-axis. As can be observed, execution time of each iteration increases after restart in case of original BLCR whereas it remains the same as before checkpointing for the affinity-aware BLCR. Table 5.2 (fourth column) depicts the percentage change in the average execution time per iteration after restart for 16 threads and Table 5.3 (fourth column) for 8 threads.

We observe that MG, LU, CG and UA show higher improvement in the application execution time after restart compared to other NAS benchmarks. They show an average reduction of more than 60% in the average execution time per iteration and int the number of CPU migrations after restart. Although BT, SP and FT show large reductions in CPU migrations, the overall performance improvement in execution time after restart is comparatively smaller. This can be attributed to an interesting observation from the second and the fourth columns of Table 5.2 and Table 5.3 : The percentage change in the application execution time after restart (second column) is equal to the percentage change in the average execution time per iteration after

Table 5.2: % Change using the affinity-aware BLCR over the original BLCR for 16 Threads

NAS benchmarks	% Change in application execution time after restart	% Change in CPU migrations after restart	% Change in average execution time per iteration after restart	% Change in checkpoint file size	% Change in checkpoint time	% Change in restart time
BT	46.35%	93.79%	46.38%	-0.13%	-0.31%	-1.33%
SP	49.23%	71.78%	49.12%	-0.12%	-1.74%	-0.14%
FT	48.49%	81.41%	48.50%	-0.02%	-0.20%	-0.20%
MG	73.95%	86.42%	73.97%	-0.04%	-10.95%	0.14%
LU	71.97%	80.07%	72.12%	-0.16%	-11.68%	-10.57%
CG	70.10%	61.08%	70.03%	-0.10%	-0.81%	0.02%
UA	64.52%	63.95%	64.64%	-0.20%	-3.90%	-1.82%
IS	37.70%	27.70%	37.32%	-0.07%	-0.85%	-0.12%

restart (fourth column). An application showing higher improvement in the average execution time per iteration also shows higher improvement in the execution time as a whole after restart. With this observation, we can infer (by process of elimination) that reduction in the average execution time per iteration after restart (due to reduction in remote memory references) has a major impact on the application performance compared to reduction in CPU migrations after restart.

5.4 Overheads

Let us consider some of the overheads one would expect in affinity-aware BLCR.

5.4.1 Checkpoint file size overhead

Figure 5.4 depicts the checkpoint file size in MB on the y-axis and the NAS benchmarks on the x-axis. Results are shown for both 16 threads (Figure 5.4(a)) and 8 threads (Figure 5.4(b)) and are almost same for both the original BLCR and the affinity-aware BLCR. Table 5.2 (fifth column) shows the percentage change in the checkpoint file size for 16 threads and Table 5.3 (fifth column) shows the results for 8 threads. Negative percentages indicate overheads incurred. We observe a size difference of around 1 MB between the checkpoint file with the original BLCR vs. the checkpoint file with the affinity-aware BLCR. The difference in the file size is due to the

Table 5.3: % Change using the affinity-aware BLCR over the original BLCR for 8 Threads

NAS benchmarks	**% Change in application execution time after restart**	**% Change in CPU migrations after restart**	**% Change in average execution time per iteration after restart**	**% Change in checkpoint file size**	**% Change in checkpoint time**	**% Change in restart time**
BT	26.84%	76.25%	26.85%	-0.07%	-1.29%	-0.17%
SP	41.05%	89.33%	41.06%	-0.06%	-1.34%	-0.23%
FT	22.71%	48.10%	22.72%	-0.01%	-0.12%	-0.19%
MG	46.22%	57.54%	46.15%	-0.02%	-6.11%	-0.12%
LU	34.83%	86.63%	34.82%	-0.08%	-1.63%	-1.23%
CG	37.36%	68.91%	37.44%	-0.05%	-0.51%	-0.58%
UA	43.99%	86.42%	43.97%	-0.11%	-1.48%	-0.34%
IS	18.72%	16.78%	18.50%	-0.04%	0.39%	-0.26%

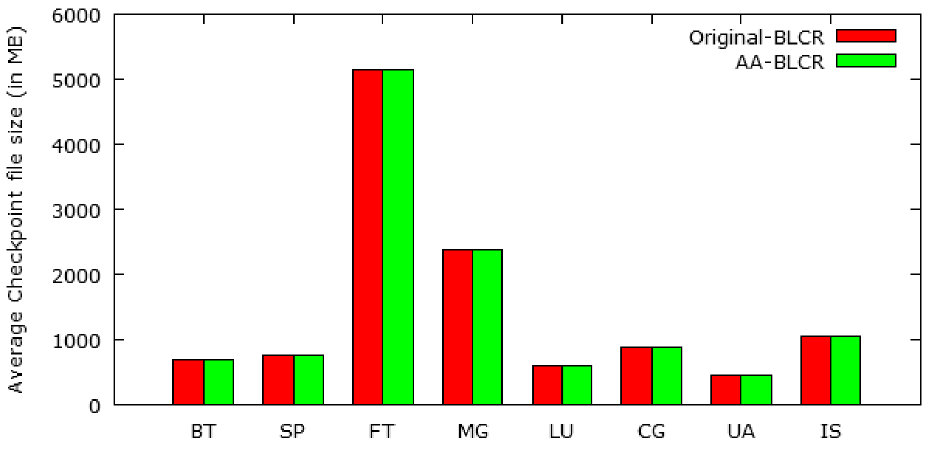
affinity-aware BLCR storing with each of the threads some meta information, including virtual address maps, protection flags, start and end marker and other information pertinent to the BLCR framework. A difference of 1MB is not significant considering the checkpoint file sizes in the 100s of MBs or even GBs.

5.4.2 Checkpoint time overhead

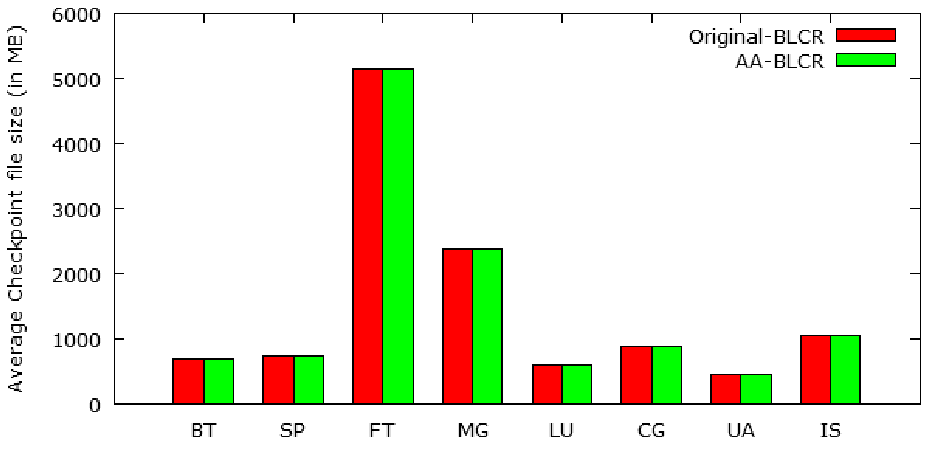
Figure 5.5 depicts the checkpoint time for the original BLCR and the affinity-aware BLCR. Figure 5.5(a) depicts results for 16 threads and Figure 5.5(b) for 8 threads. The y-axis denotes the checkpoint time in seconds and the x-axis denotes the NAS benchmarks. Table 5.2 (sixth column) shows the percentage change in the checkpoint time for 16 threads and Table 5.3 (sixth column) shows the results for 8 threads. As can be observed, the difference between the checkpoint time is also not significant.

5.4.3 Restart time overhead

Figure 5.6 depicts the restart time for the original BLCR and the affinity-aware BLCR. Figure 5.6(a) depicts results for 16 threads and Figure 5.6(b) for 8 threads. The y-axis denotes the restart time in seconds and the x-axis denotes the NAS benchmarks. Table 5.2 (seventh column) shows the percentage change in the restart time for 16 threads and Table 5.3 (seventh column) shows the results for 8 threads. Similar to the checkpoint time overhead, this difference

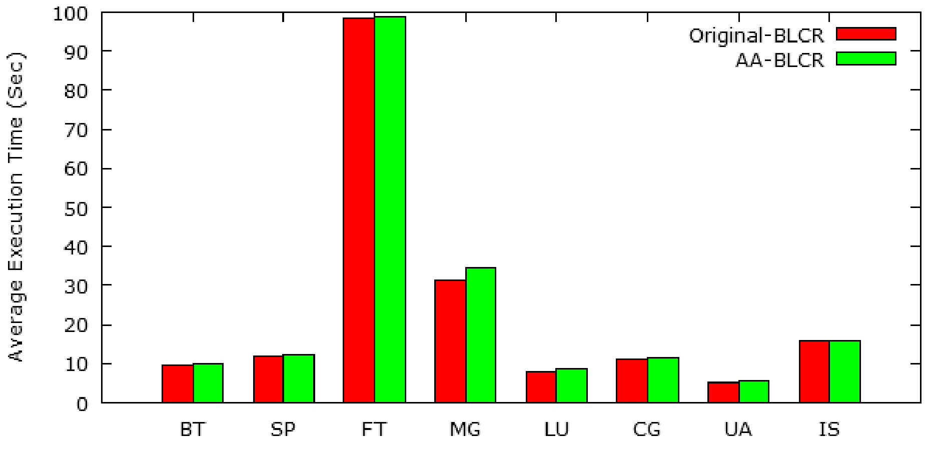


(a) 16 Threads

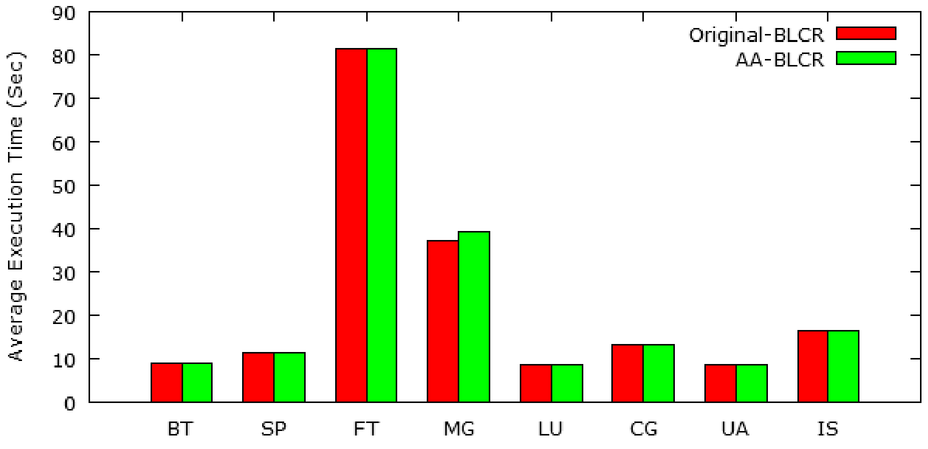


(b) 8 Threads

Figure 5.4: NAS Benchmark (CLASS C) checkpoint file size

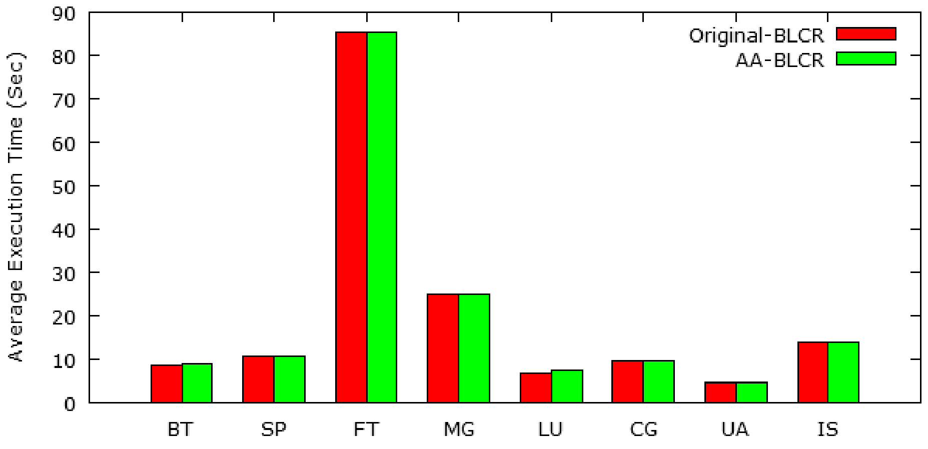


(a) 16 Threads

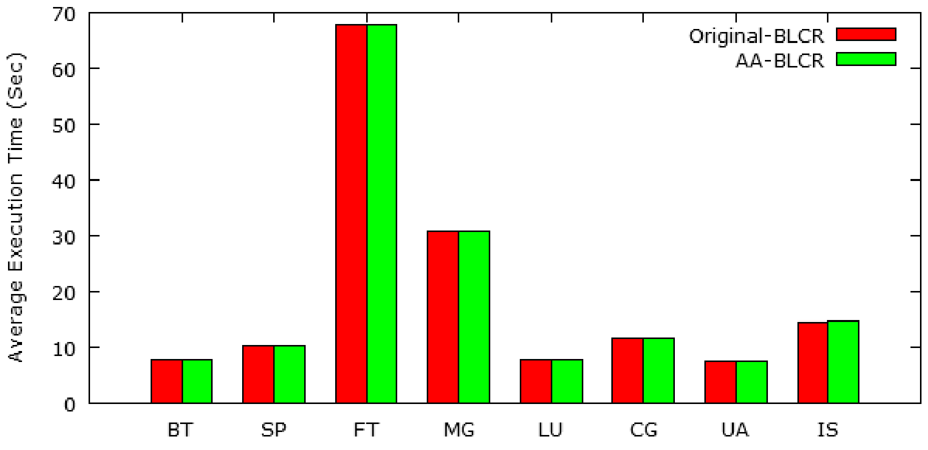


(b) 8 Threads

Figure 5.5: NAS Benchmark (CLASS C) checkpoint time



(a) 16 Threads



(b) 8 Threads

Figure 5.6: NAS Benchmark (CLASS C) restart time

Table 5.4: Results for LULESH Benchmark for 16 Threads

Input Size	**Total iterations**	**Average execution time per iteration (in sec) without pinning**	**Total application execution time (in sec) without pinning**	**Average execution time per iteration (in sec) with pinning**	**Total application execution time (in sec) with pinning**	**% Change in application execution time after restart for the affinity-aware BLCR over the original BLCR**
200	100	5.17	517.33	5.06	505.85	3.81%
200	200	5.22	1044.92	5.08	1015.91	4.56%

is also not significant.

Overall, the application execution time after restart and the overheads in terms of the percentage change in Table 5.2 and Table 5.3 show that we obtain a significant performance improvement in application execution time with affinity-aware BLCR compared to the original BLCR with only minimum overheads.

5.5 LULESH Benchmark Results

We also investigated the impact of affinity-aware BLCR on benchmarks that might not be sensitive to affinity information and/or that might actually suffer in performance when threads are pinned to specific CPU cores. In some cases, binding/pinning can degrade performance by inhibiting the OS capability to balance loads.

We conducted experiments with the LULESH benchmark. In our experiments, we instrumented LULESH to initiate a checkpoint at the 10th iteration. Table 5.4 depicts the results for 16 threads and Table 5.5 for 8 threads. The first and the second columns show the input parameters for LULESH, input size and total iterations, respectively. The third column shows the average execution time per iteration (in seconds) and the fourth column shows the total application execution time (in seconds). In both cases, thread-to-core pinning is not enforced and no checkpoint/restart is initiated. The fifth and the sixth columns show measurements similar to third and fourth column, respectively, but with thread-to-core pinning enforced. The seventh column shows the percentage change in the application execution time after restart when using the original BLCR and the affinity-aware BLCR.

When running LULESH with 16 threads, with and without thread-to-core pinning and no checkpoint restart, pinning showed only marginal benefits. This can be observed by comparing

Table 5.5: Results for LULESH Benchmark for 8 Threads

Input Size	Total iterations	Average execution time per iteration (in sec) without pinning	Total application execution time (in sec) without pinning	Average execution time per iteration (in sec) with pinning	Total application execution time (in sec) with pinning	% Change in application execution time after restart for the affinity-aware BLCR over the original BLCR
200	100	6.80	680.23	7.00	700.16	-1.59%
200	200	6.85	1369.38	7.00	1398.03	-2.36%

the fourth and the sixth columns of Table 5.4. When using checkpoint restart, the affinity-aware BLCR similarly showed marginal improvement over the original BLCR (Table 5.4 seventh column). When running LULESH with 8 threads, with and without thread-to-core pinning and no checkpoint restart, pinning showed lower performance as compared to no pinning. This can be observed by comparing the fourth and the sixth columns of Table 5.5. When using checkpoint restart, the affinity-aware BLCR similarly showed lower performance as compared to the original BLCR (Table 5.5 seventh column). One of the reasons for such performance results is that LULESH performs memory allocation and de-allocation dynamically in each iteration. Hence, the data used in one iteration is more or less independent of the data used in earlier iterations. This is also evident in the average execution time per iteration, that changes only slightly, irrespective of whether or not the threads were pinned. This is observed by comparing the third and the fifth columns of Table 5.4 for 16 threads and Table 5.5 for 8 threads.

Based on these experiments, we conclude that applications sensitive to thread-to-core pinning and page-to-NUMA node mappings will obtain significant benefits when using the affinity-aware BLCR. But for other applications that are not sensitive to such mappings, performance improvement depends on how they themselves perform with and without pinning. If they show some benefits with pinning, the affinity-aware BLCR will also show benefits. Otherwise, one should revert to using the original BLCR for such applications. To this end, including affinity awareness can be enabled/disabled by command line arguments and environment variables.

Chapter 6

Related Work

Checkpoint and restart as a tool for fault tolerance has been well studied. There are several implementations of checkpoint restart mechanisms, including user-level implementations, kernel-level implementations, hybrid implementations. Some of the implementations support incremental checkpointing, some implement coordinated and uncoordinated checkpoint-restart techniques, some support checkpointing only single-threaded applications whereas others support checkpointing multi-threaded applications. [22] provides a survey of different checkpointing techniques.

DMTCP [6] (Distributed MultiThreaded CheckPointing) is a transparent user-level checkpointing package for distributed applications. It can checkpoint multi-threaded applications. CryoPID [9] is an open source user-level implementation, which consists of a program called freeze that captures the state of a running process and writes it into a file. CRAK [30] is a transparent checkpoint/restart kernel module for Linux. But CRAK cannot restart multithreaded processes since it does not capture shared virtual memory areas. BLCR [11], which we have used in our implementation, is a hybrid checkpoint restart mechanism providing a loadable Linux kernel module.

Significant research has been conducted to lower the overheads in checkpoint restart mechanisms. Oliner et al. [21] present a cooperative checkpointing approach that reduces overheads by only writing checkpoints that are predicted to be useful, e.g., when a failure is likely in the near future. Incremental checkpointing [5], [18] reduces the size of full checkpoints taken by periodically saving changes in the application data between full checkpoints. Moody et al. [17] discusses the design and modeling of a scalable multi-level checkpointing system and recent work [26] uses a combination of non-blocking and multi-level checkpointing. [13] discusses an uncoordinated checkpointing protocol for send deterministic MPI HPC applications. A given MPI application is said to be send deterministic, if, for a set of input parameters, the sequence of sent messages, for any process, is the same in any correct execution. AI-Ckpt [20] provides a runtime environment that enables asynchronous incremental checkpointing. Unlike other C/R approaches, it leverages both current and past access pattern trends in order to optimize the order in which memory pages are flushed to stable storage. Scalable Pattern-Based Checkpointing (SPBC) [27] is a protocol that combines hierarchically coordinated checkpointing and message logging. Libhashckpt [12] is a hybrid incremental checkpointing solution that utilizes both page protection and hashing on GPUs to determine changes in application data with very low overhead. ACR [19] is an automatic checkpoint/restart framework that performs application replication and automatically adapts the checkpoint period exploiting online information about the current failure rate. Sarood et al. [25] discuss a combination of checkpoint/restart and temperature capping. It uses a runtime managed temperature capping to increase the estimated reliability of HPC machines and reduce the total execution time required by applications. Algorithm-based fault tolerance (ABFT) techniques [8], [16] provide a solution for HPC resilience to applications.

We investigated several of these existing checkpoint restart implementations, but, to the best of our knowledge, none of these implementations provide affinity awareness as we have described in this work.

Chapter 7

Future Work and Conclusion

7.1 Future Work

This work is part of a project to extend BLCR to provide resilience support for Partitioned Global Address Space (PGAS) environments [23], [24]. PGAS is the programming paradigm in the DEGAS [29] stack, which aims to develop the next generation of programming models, runtime systems and tools to meet the challenges of Exascale systems. PGAS provides a partitioned (data designated as local or global) global (capability to directly read/write remote data) view of address space. It provides affinity control through allocating data on local or global address spaces. However, as we have described in our work, the existing implementation of BLCR does not restore affinity information across restarts. We have already incorporated affinity awareness in BLCR and we will integrate this capability into the PGAS environment of DEGAS.

7.2 Conclusion

In conclusion, this work contributes a novel approach to incorporate affinity awareness in a checkpoint restart mechanism. We have implemented our design in BLCR with minimal changes and minimal overheads. Experimental results with the NAS benchmark suite indicate significant performance benefits over the original BLCR. Affinity-aware BLCR is bound to result in benefits for affinity sensitive application but, at the same time, we also discuss an example of an application that is not sensitive to affinity, namely LULESH. As core pinning does not provide benefits for LULESH in terms of execution time, using the affinity-aware BLCR also cannot provide performance improvement.

Experiments with the NAS PB codes and LULESH showed that we are able to retain the runtime performance of applications across restarts, which confirms the hypothesis.

REFERENCES

[1] HPC - High Performance Supercomputing. *http://insidehpc.com/hpc-basic-training/what-is-hpc/.*

[2] Introduction to Parallel Computing. *https://computing.llnl.gov/tutorials/parallel_comp/.*

[3] OpenMP. *https://computing.llnl.gov/tutorials/openMP/.*

[4] Top500. *http://www.top500.org/.*

[5] Saurabh Agarwal, Rahul Garg, Meeta S Gupta, and Jose E Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 277–286. ACM, 2004.

[6] Jason Ansel, Kapil Arya, and Gene Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[7] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Russell L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[8] Zizhong Chen. Online-abft: an online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–176. ACM, 2013.

[9] cryopid-devel@lists.berlios.de. CryoPID - a process freezer for linux. *https://github.com/maaziz/cryopid*, 2004.

[10] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing numa locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles*

and Practice of Parallel Programming, PPoPP '12, pages 247–256, New York, NY, USA, 2012. ACM.

[11] Jason Duell. The design and implementation of berkeley lab's linux checkpoint/restart. 2005.

[12] Kurt B Ferreira, Rolf Riesen, Ron Brighwell, Patrick Bridges, and Dorian Arnold. libhashckpt: hash-based incremental checkpointing using gpus. In *Recent Advances in the Message Passing Interface*, pages 272–281. Springer, 2011.

[13] Amina Guermouche, Thomas Ropars, Elisabeth Brunet, Marc Snir, and Franck Cappello. Uncoordinated checkpointing without domino effect for send-deterministic mpi applications. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 989–1000. IEEE, 2011.

[14] Haoqiang Jin, Michael Frumkin, and Jerry Yan. The openmp implementation of nas parallel benchmarks and its performance. Technical report, Technical Report NAS-99-011, NASA Ames Research Center, 1999.

[15] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, et al. Exploring traditional and emerging parallel programming models using a proxy application. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 919–932. IEEE, 2013.

[16] Dong Li, Zizhong Chen, Panruo Wu, and Jeffrey S Vetter. Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[17] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *High Perfor-*

mance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for, pages 1–11. IEEE, 2010.

[18] Nichamon Naksinehaboon, Yudan Liu, Chokchai Leangsuksun, Raja Nassar, Mihaela Paun, and Stephen L Scott. Reliability-aware approach: An incremental checkpoint/restart model in hpc environments. In *Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on*, pages 783–788. IEEE, 2008.

[19] Xiang Ni, Esteban Meneses, Nikhil Jain, and Laxmikant V Kalé. Acr: automatic checkpoint/restart for soft and hard error protection. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 7. ACM, 2013.

[20] Bogdan Nicolae and Franck Cappello. Ai-ckpt: leveraging memory access patterns for adaptive asynchronous incremental checkpointing. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 155–166. ACM, 2013.

[21] Adam J Oliner, Larry Rudolph, and Ramendra K Sahoo. Cooperative checkpointing: a robust approach to large-scale systems reliability. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 14–23. ACM, 2006.

[22] Eric Roman. A survey of checkpoint/restart implementations. In *Lawrence Berkeley National Laboratory, Tech.* Citeseer, 2002.

[23] Eric Roman. Overview of degas programming models area. Technical report, DEGAS Summer Retreat, LBNL, 2013.

[24] Vivek Sarkar. Resilient runtimes for global address languages. Technical report, DEGAS Summer Retreat, LBNL, 2013.

[25] Osman Sarood, Esteban Meneses, and Laxmikant V Kale. A cool way of improving the reliability of hpc machines,. In *Proceedings of The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013.

[26] Kento Sato, Naoya Maruyama, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R de Supinski, and Satoshi Matsuoka. Design and modeling of a non-blocking checkpointing system. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 19. IEEE Computer Society Press, 2012.

[27] André Schiper, Franck Cappello, Tatiana Martsinkevich, Amina Guermouche, and Thomas Ropars. Spbc: Leveraging the characteristics of mpi hpc applications for scalable checkpointing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC" 13)*, number EPFL-CONF-189836, 2013.

[28] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Hybrid checkpointing for mpi jobs in hpc environments. In *Proceedings of the 2010 IEEE 16th International Conference on Parallel and Distributed Systems*, ICPADS '10, pages 524–533, Washington, DC, USA, 2010. IEEE Computer Society.

[29] Katherine Yelick, Vivek Sarkar, James Demmel, Mattan Erez, Dan Quinlan, Surendra Byna, Paul Hargrove, Steven Hofmeyr, Costin Iancu, Khaled Ibrahim, Leonid Oliker, Eric Roman, John Shalf, David Skinner, Erich Strohmaier, Samuel Williams, and Yili Zheng. Degas: Dynamic exascale global address space. Technical report, DEGAS Summer Retreat, LBNL, 2013.

[30] Hua Zhong and Jason Nieh. Crak: Linux checkpoint/restart as a kernel module. Technical report, Technical Report CUCS-014-01, Department of Computer Science, Columbia University, 2001.