Affinity-Aware Checkpoint Restart

Ajay Saini, Arash Rezaei, Frank Mueller
North Carolina State University
Raleigh, NC
mueller@ncsu.edu

Paul Hargrove, Eric Roman
Lawrence Berkeley National Laboratory
Berkeley, CA
{phhargrove, eroman}@lbl.gov

ABSTRACT

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 today, 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 of 37%-73% for the NAS Parallel Benchmark codes and 6-12% for NAMD with negligible overheads instead of up to nearly four times longer an execution times without affinity-aware restarts on 16 cores.

Categories and Subject Descriptors

D.4.5 [**Reliability**]: Checkpoint/Restart—*Fault Tolerance*

General Terms

Reliability, Efficiency

Keywords

Checkpoint and restart, fault tolerance, multi-core, NUMA, system software

1. INTRODUCTION

With the recent rise in the number of processing cores on HPC systems (10,000s or even 100,000s of processing cores), faults are becoming common. The mean time between failures (MTBF) / interrupts (MTBI) is in the range of 6.5-50 hours depending on the maturity / age of HPC installations [25]. 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 [21].

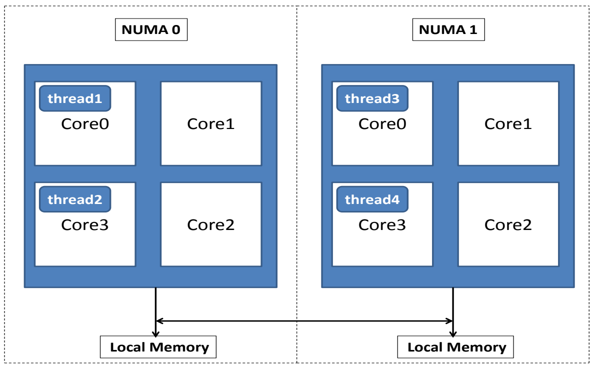
Another notable development attributed to the increase in the number of cores is an accelerated shift towards distributed non-uniform memory access (NUMA) architectures. Such architectures consist of collections of computing cores with fast local memory, communicating with each other via a slower inter-chip 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 their 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 1(a) shows an application with 4 threads pinned to particular cores. But when the same application is executed without pinning (Figure 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.

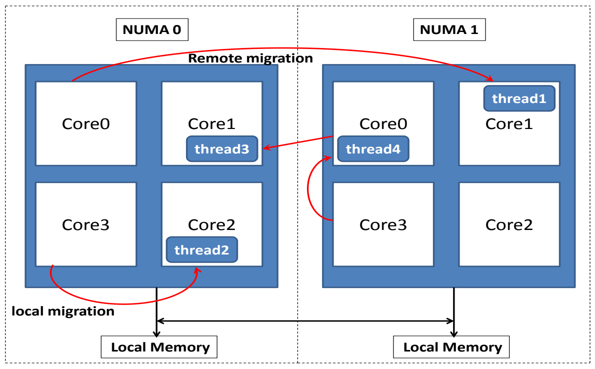
There are various real world scenarios where pinning is beneficial, especially for applications which are sensitive to such placement. For example, Dice et. al [6] talk about the benefits of NUMA aware locks. Even Operating System (OS) process schedulers have inbuilt intelligence to reduce migrations, 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 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 threads pinned as in Figure 1(a),

Middleware '14 December 08 – 12 2014, Bordeaux, France
ACM 978-1-4503-2785-5/14/12 ...$15.00.
http://dx.doi.org/10.1145/2663165.2663325.



(a) Application threads pinned to CPU cores



(b) Application threads not pinned resulting in migrations

Figure 1: Effects of pinning and no-pinning

when such an application is restarted from a checkpoint, we might end up with thread-to-core mapping as in Figure 1(b) since pinning is not preserved. This is the problem we target in this paper: How can we ensure that affinity information is preserved across restarts?

Contributions

When we restart an application from a checkpoint, we want that application to exhibit the same affinity behavior it had before the checkpoint. We present a novel approach to save and restore affinity information. We have implemented our design in BLCR [7], enhancing it to affinity-aware BLCR. BLCR 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. An evaluation of the benefits of our enhancements shows performance improvements ranging from 37% to 73% in application execution time for NAS Parallel Benchmark (NPB) and 6-12% for NAMD after restart compared to using the original BLCR on 16 cores. Without affinity awareness, restarts would have resulted in up to nearly four times longer execution times. To the best of our knowledge, we are first to implement such affinity awareness in a checkpoint restart mechanism.

The paper is organized as follows: Section II discusses the design of the original BLCR and presents our affinity-aware design. Section III provides the implementation details. In Section IV, we discuss our results. Section V compares this work to related work. Section VI presents our conclusions.

2. 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 [7].

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. Figures 2 and 3 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 affinity aware BLCR. We use a running example of an application with three threads in our description.

2.1 Original BLCR Checkpoint-Restart Flow

2.1.1 Checkpoint flow

When a checkpoint request is triggered, it results in the following sequence of actions (see Figure 2):

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.

2.1.2 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):

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 then remaps them based on the information stored in the checkpoint file. It loads the shared items including dirty pages and uses kernel support (*sys_mprotect* in the 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 reload 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.

2.2 Saving and Restoring Affinity Information

The checkpoint and restart flow described in the previous section does not consider affinity information, i.e., thread-to-core and page-to-NUMA node mappings 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 extract and save the cpumask for each thread during checkpointing. During restart, we 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 (during restart). 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 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 node id 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. Furthermore, 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, today, are NUMA aware, i.e., their memory allocator assigns pages to the local NUMA node where the thread is running using the first touch policy [14]. This approach does not inflict migration overheads. We used the second approach. The first approach was prototyped at the design level and then discarded for two reasons:

- It required duplication of significant Linux kernel code snippets with slight variations, which would have made it harder to maintain the code as the Linux kernel is changed in the future.
- All data would first be located on one NUMA node of the first thread upon restart before being migrated to their original NUMA nodes. This migration incurs additional overhead compared to our second approach where pages are placed on their original NUMA node right away when loaded.

2.3 Affinity-Aware BLCR 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.

2.3.1 Checkpoint flow

A checkpoint request results in the following actions (see Figure 2):

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.

2.3.2 Restart flow

A restart request results in the following actions (see Figure 3):

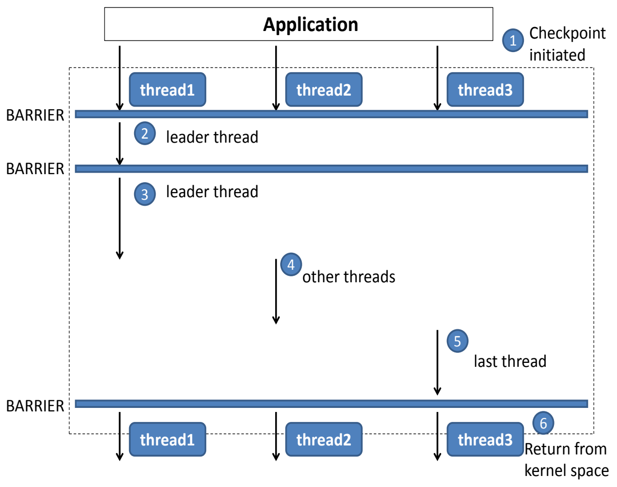
Step1: The same as earlier.

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.

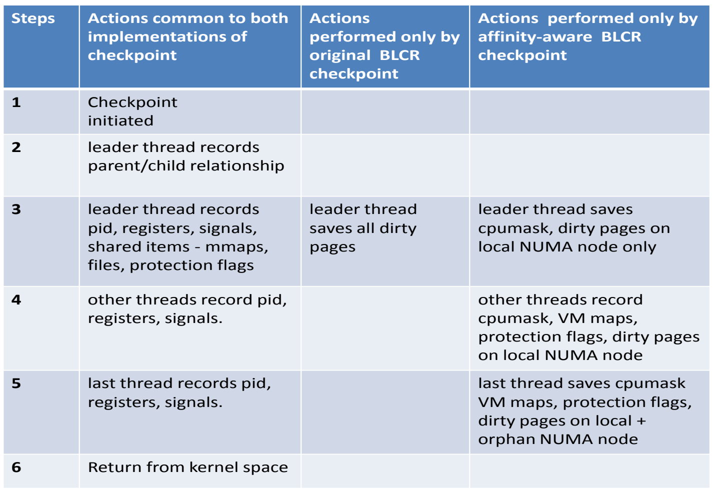
Step3: The other threads load their private data, including registers, signals and cpumask. Kernel support is needed to reset the cpumask. Then, the saved pages are loaded, and the pages are allocated on the correct NUMA nodes.

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 checkpointing, for each of the virtual memory maps.

Step5 and Step6: The same as earlier.



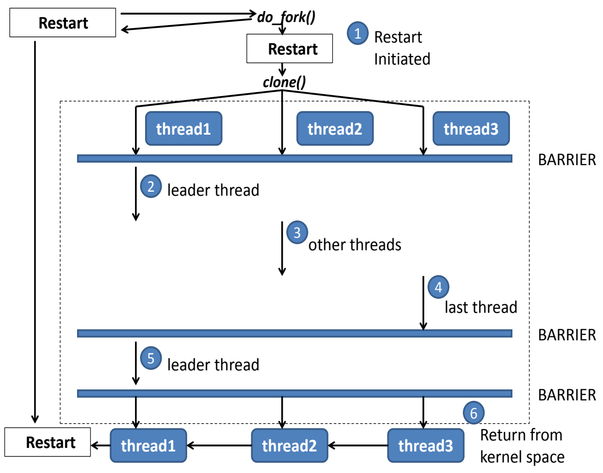
(a) Checkpoint flow diagram



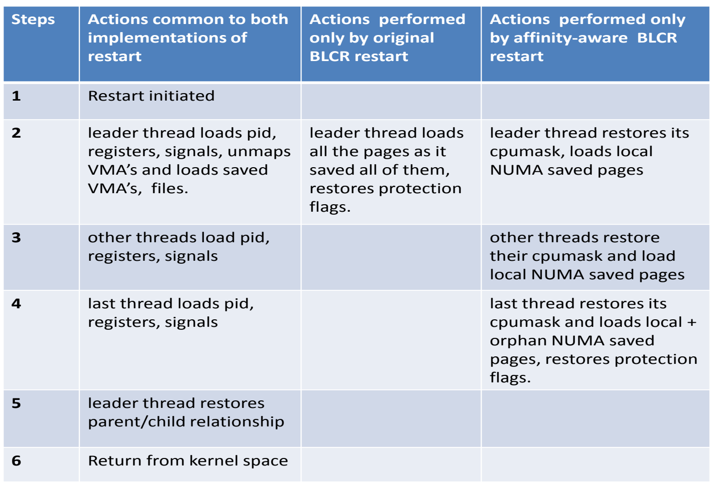
Steps	Actions common to both implementations of checkpoint	Actions performed only by original BLCR checkpoint	Actions performed only by affinity-aware BLCR checkpoint
1	Checkpoint initiated		
2	leader thread records parent/child relationship		
3	leader thread records pid, registers, signals, shared items - 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 2: 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: Restart flow

3. 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 a minimum. *Third*, new updates should be flexible, i.e., they can be turned on/off via command line parameters and/or environment variables.

3.1 Saving and Restoring Thread-to-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_ptr(
        struct task_struct *p,
        const struct cpumask *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.

3.2 Saving and Restoring Page-to-NUMA Node Affinity

To save and restore page-to-NUMA node affinity, we rely on the first touch policy of Linux memory allocator, which assigns 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 mappings, we need page table information. BLCR already implements a page table walk 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, should the user desire such different mappings. This might be desirable if mixed workloads are run on nodes so that only different cores are available on restart.

4. RESULTS

4.1 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 [3], [10] (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 provide results for a real world application, NAMD [20], which was reported to result in longer runtimes after restarts by a user of BLCR. NAMD is a parallel molecular dynamics code designed for high-performance simulation of large bio-molecular systems. It scales to hundreds of processors on high-end parallel platforms as well as tens of processors on low-cost commodity clusters, and also runs on individual desktop and laptop computers. It is implemented in C++ and based on Charm++ parallel objects [11].

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 the LULESH [12] benchmark to be one such example. LULESH 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.

4.2 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 NPB codes 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 representative of HPC workloads. 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 NPB codes to initiate a checkpoint at a particular iteration count after initialization as shown in Table 1. The first column in Table 1 lists the NPB codes, the second column shows the total iterations in the complete run of that benchmark and the third column shows the iteration number after which the checkpoint was initiated. In selecting an iteration to checkpoint at, we tried to ensure a sufficiently long execution time after restart. This allowed us to assess if execution time varies after restart over time due to affinity effects. Our experiments actually show that no such variation was observed, i.e., restarting from checkpoint resulted in similar performance irrespective of the remaining work.

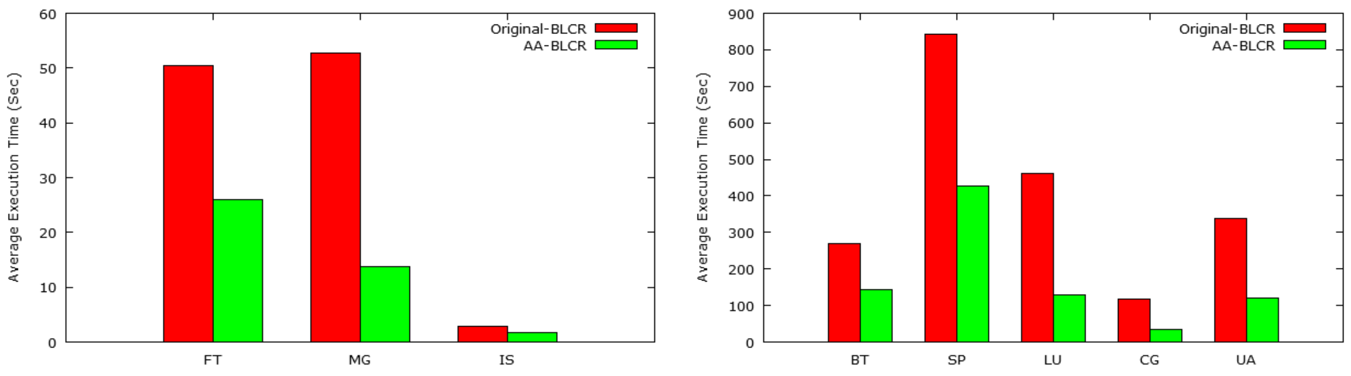
During experiments with each benchmark, we took five different checkpoints at the same iteration count. Each checkpoint was restarted two times. This was done both for the original BLCR and the affinity-aware BLCR. The results presented next report average values of these runs. The percentage change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

$$\frac{(OriginalBLCR\ Time\ -\ AffinityAwareBLCR\ Time)}{OriginalBLCR\ Time} \times 100$$

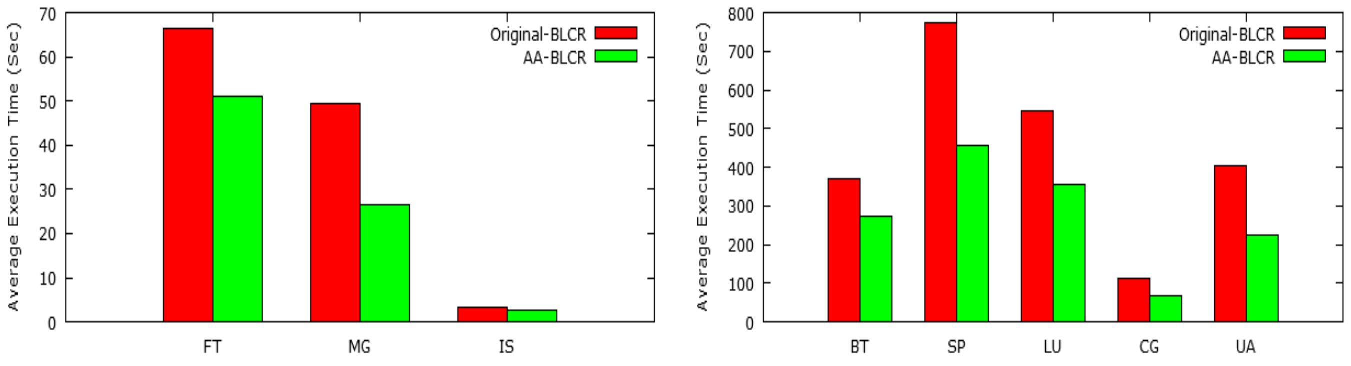
The standard deviation of the results presented in this section is less than 3% except for one case. For this case, we depict standard deviation as min/max values through error bars in graph (see below).

4.3 Performance

Figure 4 depicts the execution time of the NPB codes after restart from a checkpoint file using the original BLCR and the affinity-aware BLCR (AA-BLCR). Figure 4(a) depicts results when each of the NPB codes are configured to run with 16 threads. Figure 4(b) depicts results for 8 threads.

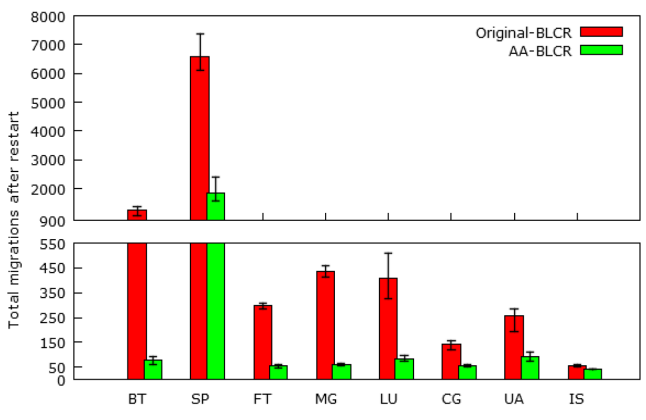


(a) 16 Threads

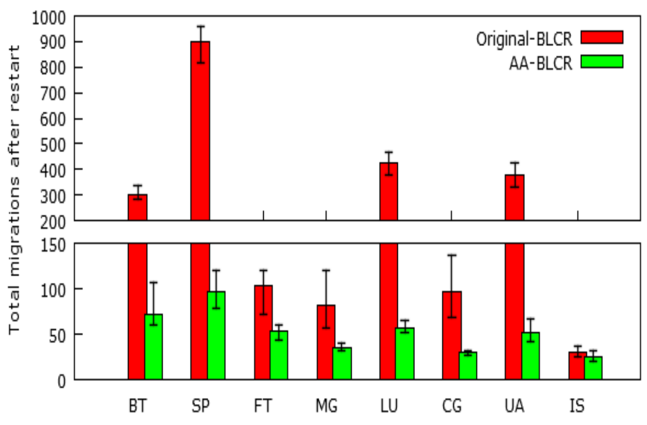


(b) 8 Threads

Figure 4: NPB codes (CLASS C) Execution Time after Restart (excl. Restart Time)



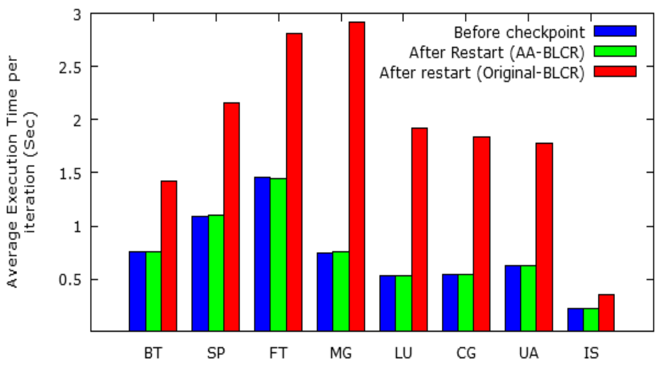
(a) 16 Threads



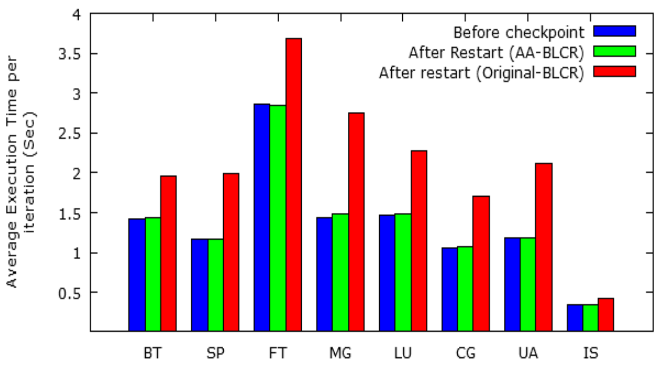
(b) 8 Threads

Figure 5: NPB codes (CLASS C) # CPU Migrations after Restart

The x-axis depicts each of the NPB codes 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. Benchmarks FT, MG, IS are depicted in the left half and BT, SP, LU, CG, UA are depicted in the right halves of Figures 4(a) and 4(b). Table 2 depicts the percentage change between the original BLCR and the affinity-aware BLCR for different parameters using 16 threads. The first column lists the NPB codes and the second column shows the percentage change in the application execution time after restart. Table 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 2 second column) and between 18% and 46% for 8 threads (Table 3 second column). For 16 threads, this means



(a) 16 Threads



(b) 8 Threads

Figure 6: NPB codes (CLASS C) Avg. Execution Time per Iteration before+after Checkpoint for AA-BLCR / Original-BLCR

Table 1: Total # Iterations / Last Iteration before Checkpoint

NPB	Total # Iterations	Checkpoint Iteration
BT	200	10
SP	400	10
FT	20	2
MG	20	2
LU	250	10
CG	75	10
UA	200	10
IS	10	2

applications would result in roughly 1.6 times to nearly four times longer execution than before checkpointing unless the restart was affinity aware.

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 Figures 5 and 6. Figure 5 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 NPB code on the x-axis. The min/max values for each of these cases are shown as error bars. The *migrations* event of the *perf stat* command was used to obtain CPU migration numbers. Figure 5(a) depicts results for 16 threads and Figure 5(b) for 8 threads. Due to the large range in the data values, the y-axis is divided into two intervals as shown in Figures 5(a) and 5(b). We observe a large reduction in CPU migrations for the affinity-aware BLCR as thread-to-core maps are restored. Table 2 (third column) depicts the percentage change in CPU migrations for 16 threads and Table 3 (third column) for 8 threads. There are still some migrations that are attributed to migrations before thread-to-core maps are restored.

Figure 6 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 NPB codes on the x-axis. As can be observed, execution time of each iteration increases after restart in case of the original BLCR whereas it remains the same as before checkpointing for the affinity-aware BLCR. Table 2 (fourth column) depicts the percentage change ranging from 37%-73% in the average execution time per iteration after restart for 16 threads and Table 3 (fourth column) for 8 threads. This performance change is the most significant result as it is independent of the time at which a checkpoint is taken, i.e., the amount of saved execution time can be extrapolated if the remaining number of iterations is known.

We observe that MG, LU, CG and UA show higher improvement in the application execution time after restart compared to other NPB codes. They show an average reduction of more than 60% in the average execution time per iteration and in 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 Tables 2 and 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 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 the reduction in the average execution time per iteration after restart (due to reduction in remote memory references) has a dominant impact on the application performance compared to a lower impact due to the reduction in CPU migrations after restart.

4.4 Overheads

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

4.4.1 Checkpoint file size

Figure 7 depicts the checkpoint file size in MB on the y-axis and the NPB codes on the x-axis. Results are shown for 16 threads (Figure 7) and are nearly the same for 8 threads

Table 2: % Change Affinity-Aware BLCR vs. Original BLCR for 16 Threads

NPB	change app time after restart (%)	change CPU migrations after restart (%)	change avg. time / iteration after restart (%)	change check-point file size (%)	change check-point time (%)	change 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

Table 3: % Change Affinity-Aware BLCR over Original BLCR for 8 Threads

NPB	change app time after restart (%)	change CPU migrations after restart (%)	change avg. time / iteration after restart (%)	change check-point file size (%)	change check-point time (%)	change 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

(graph omitted) since each thread only adds a few KB data to traces of many MBs. We observe that file sizes are almost the same for both the original BLCR and the affinity-aware BLCR. Table 2 (fifth column) shows the percentage change in the checkpoint file size for 16 threads and Table 3 (fifth column) shows the results for 8 threads. In this case, percentages indicate overheads incurred by our method. 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 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 that checkpoint file sizes are in the order of 100s of MBs or even GBs.

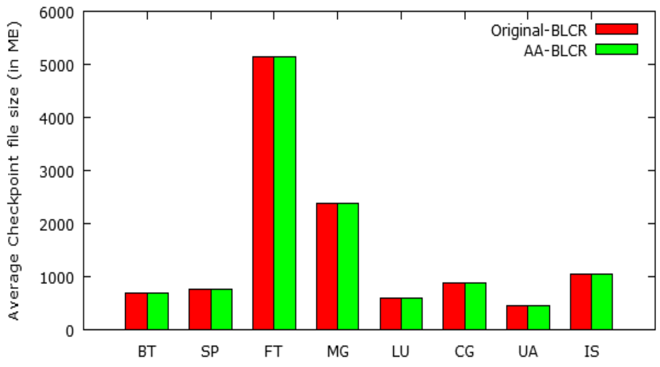


Figure 7: NPB codes (CLASS C) Checkpoint File Size for 16 Threads

4.4.2 *Checkpoint time overhead*

Table 4: NAMD Executing Time [hours] for 16 Threads

config file	app time after 5000th iter w/o C/R (hrs)	app time after 5000th iter on restart AA-BLCR (hrs)	app time after 5000th iter on restart Original-BLCR (hrs)	% change time AA-BLCR vs. Original-BLCR
apoa1.namd	16.10	16.12	17.18	6.17%
npt55.inp	18.42	18.40	21.00	12.38%

Table 5: NAMD Executing Time [hours] for 8 Threads

config file	app time after 5000th iter w/o C/R (hrs)	app time after 5000th iter on restart AA-BLCR (hrs)	app time after 5000th iter on restart Original-BLCR (hrs)	% change time AA-BLCR vs. Original-BLCR
apoa1.namd	31.75	31.79	32.66	2.66%
npt55.inp	35.95	36.09	37.42	3.55%

Figure 8 depicts the checkpoint time for the original BLCR and the affinity-aware BLCR. Figure 8(a) depicts results for 16 threads and Figure 8(b) for 8 threads. The y-axis denotes the checkpoint time in seconds and the x-axis denotes the NPB codes. Table 2 (sixth column) shows the percentage change in the checkpoint time for 16 threads and Table 3 (sixth column) shows the results for 8 threads. As can be observed, the difference between the checkpoint time is also not significant.

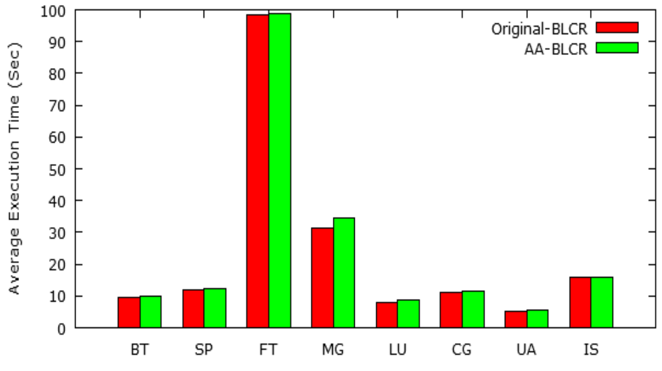
4.4.3 *Restart time overhead*

Figure 9 depicts the restart time for the original BLCR and the affinity-aware BLCR. Figure 9(a) depicts results for 16 threads and Figure 9(b) for 8 threads. The y-axis denotes the restart time in seconds and the x-axis denotes the NPB codes. Table 2 (seventh column) shows the percentage change in the restart time for 16 threads and Table 3 (seventh column) shows the results for 8 threads. Similar to the checkpoint time overhead, this difference is also not significant.

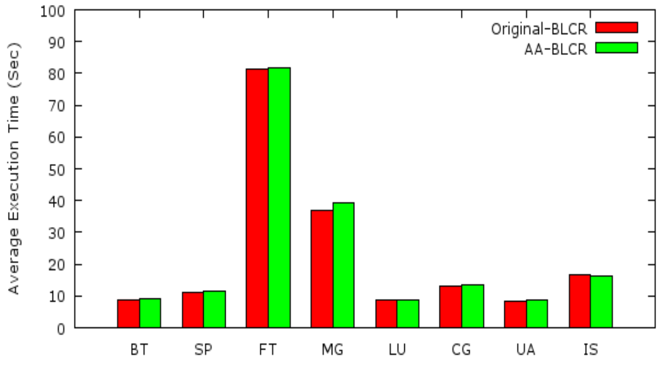
Overall, the application execution time after restart and the overheads in terms of the percentage change in Tables 2 and 3 show that we obtain a significant performance improvement in application execution time with affinity-aware BLCR compared to the original BLCR with only minimum overheads.

4.5 NAMD Results

As a case study, we investigated the benefits of using the affinity-aware BLCR over the original BLCR for the real world application NAMD. We ran NAMD in a configuration provided by a BLCR user (npt55.inp) and with a default configuration file (apoa1.namd) from the NAMD sources. The total number of iterations in each case was 500,000. We checkpointed after 1,000 iterations. These values were chosen just to match the user's configuration for NAMD and BLCR. NAMD outputs periodically (under a configurable option) how much time is left for the application to complete. We capture this value after 5,000 iterations in each case. Tables 4 and 5 present results for 16 and 8 threads, respectively. The first column of these tables show the configuration files. The second column show the time left (in hours) for the application to complete (after the 5000th iteration) without any checkpoint/restart. The third and the fourth columns represent the time left (in hours) for application to complete (after 5000th iteration) after restart using the affinity-aware BLCR and using the original BLCR, respectively. The fifth column represents the % improvement of the affinity-aware BLCR over the original BLCR. We ob-

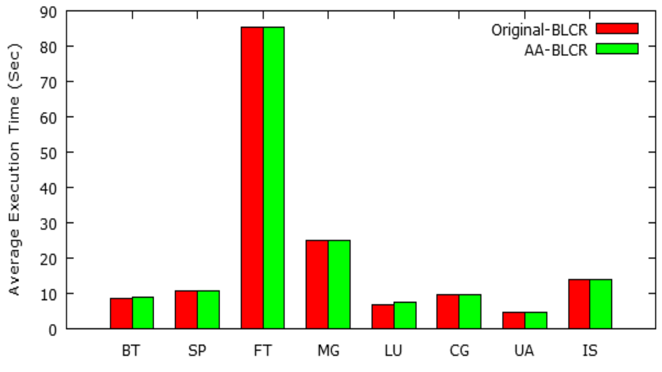


(a) 16 Threads

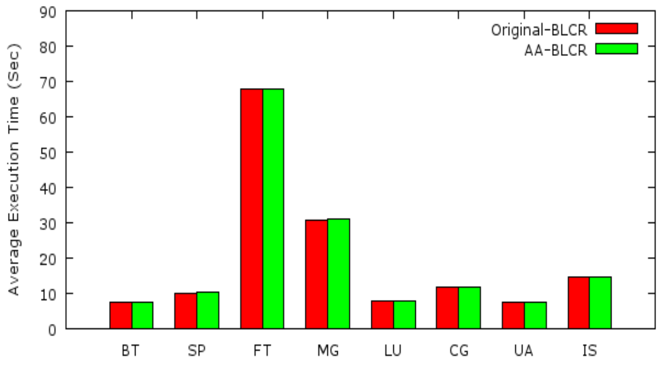


(b) 8 Threads

Figure 8: NPB codes (CLASS C) Checkpoint Time



(a) 16 Threads



(b) 8 Threads

Figure 9: NPB codes (CLASS C) Restart Time

serve that the affinity-aware BLCR (third column of Tables 4 and 5) maintains the same timing after restart as a run without any checkpoint/restart (second column of Tables 4 and 5), whereas under the original BLCR (fourth column of Tables 4 and 5), performance degrades by up to 12% after restart. Notice also that performance degradation gets worse under the original BLCR as the thread count (equal to core count, i.e., one thread per core) increases. In other words, the performance improvement by affinity-aware BLCR also increases. This can be seen in the fifth column of Tables 4 and 5 when comparing the % performance improvement for 16 and 8 threads.

4.6 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 OpenMP version of LULESH benchmark. In our experiments, we instrumented LULESH to initiate a checkpoint at the 10th iteration. Table 6 depicts the results for 16 threads and Table 7 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 the third and fourth columns, 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 but without checkpoint restart, pinning showed only marginal benefits. This can be observed by comparing the fourth and the sixth columns of Table 6. When using checkpoint restart, the affinity-aware BLCR similarly showed marginal improvement over the original BLCR (Table 6 seventh column). When running LULESH with 8 threads, with and without thread-to-core pinning and no checkpoint restart, pinning showed lower performance compared to no pinning. This can be observed by comparing the fourth and the sixth columns of Table 7. When using checkpoint restart, the affinity-aware BLCR similarly showed lower performance compared to the original BLCR (Table 7 seventh column). One of the reasons for such results is that LULESH performs memory allocation and deallocation 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

Table 6: LULESH Execution Time [secs] for 16 Threads

Input Size	total # iterations	avg. time / iteration unpinned (sec)	total app time unpinned (sec)	avg. time / iteration pinned (sec)	total app time pinned (sec)	% change in app time after restart AA-BLCR vs. 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%

Table 7: LULESH Execution Time [secs] for 8 Threads

Input Size	total # iterations	avg. time / iteration unpinned (sec)	total app time unpinned (sec)	avg. time / iteration pinned (sec)	total app time pinned (sec)	% change in app time after restart AA-BLCR vs. 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%

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 6 for 16 threads and Table 7 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 improvements depend on how they themselves perform with and without pinning. If they show some benefit with pinning, the affinity-aware BLCR will also show a benefit. 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 in our affinity-aware BLCR.

5. 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, and 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. [21] provides a survey of different checkpointing techniques.

DMTCP [2] (Distributed MultiThreaded CheckPointing) is a transparent user-level checkpointing package for distributed applications. It can checkpoint multi-threaded applications. CryoPID [5] 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 [26] 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 [7], 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. [19] 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 [1], [16] reduces the size of full checkpoints taken by periodically saving changes in the application data between full checkpoints. Moody et al. [15] discuss the design and modeling of a scalable multi-level checkpointing system and recent work [23] uses a combination of non-blocking and multi-level checkpointing. Guermouche et al. [9] discuss 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 [18] 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) [24] is a protocol that combines hierarchically coordinated checkpointing and message logging. Libhashckpt [8] 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 [17] 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. [22] 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 to reduce the total execution time required by applications. Algorithm-based fault tolerance (ABFT) techniques [4], [13] 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 paper.

6. 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 NPB suite indicate significant performance benefits over the original BLCR. The affinity-aware BLCR is bound to result in benefits for affinity sensitive application. 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. Overall, we observe performance improvements ranging from 37% to 73% in application execution time after restart for NPB codes and 6-12% for NAMD compared to using the original BLCR on 16 cores. Without affinity awareness, restarts would have resulted in 1.6 times to nearly four times longer execution times for NPB. To the best of our knowledge, we are first to implement such affinity awareness in a checkpoint restart mechanism.

Acknowledgment

This work was supported in part by grants from Lawrence Berkeley National Laboratory and NSF grant 1058779.

7. REFERENCES

[1] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. Adaptive incremental checkpointing for massively

parallel systems. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 277–286. ACM, 2004.

[2] J. Ansel, K. Arya, and G. 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.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[4] Z. 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.

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

[6] D. Dice, V. J. Marathe, and N. 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.

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

[8] K. B. Ferreira, R. Riesen, R. Brighwell, P. Bridges, and D. Arnold. libhashckpt: hash-based incremental checkpointing using gpuâĂŹs. In *Recent Advances in the Message Passing Interface*, pages 272–281. Springer, 2011.

[9] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. 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.

[10] H. Jin, M. Frumkin, and J. Yan. The openmp implementation of nas parallel benchmarks and its performance. Technical report, Technical Report NAS-99-011, NASA Ames Research Center, 1999.

[11] L. V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 91–108, 1993.

[12] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. 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.

[13] D. Li, Z. Chen, P. Wu, and J. 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.

[14] J. Marathe and F. Mueller. Hardware profile-guided automatic page placement for ccNUMA systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 90–99, Mar. 2006.

[15] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

[16] N. Naksinehaboon, Y. Liu, C. Leangsuksun, R. Nassar, M. Paun, and S. 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.

[17] X. Ni, E. Meneses, N. Jain, and L. 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.

[18] B. Nicolae and F. 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.

[19] A. J. Oliner, L. Rudolph, and R. 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.

[20] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten. Scalable molecular dynamics with namd. *Journal of computational chemistry*, 26(16):1781–1802, 2005.

[21] E. Roman. A survey of checkpoint/restart implementations. In *Lawrence Berkeley National Laboratory, Tech.* LBNL, 2002.

[22] O. Sarood, E. Meneses, and L. V. Kale. A âĂIJcoolâĂİ way of improving the reliability of hpc machines,âĂİ. In *Proceedings of The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013.

[23] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. 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.

[24] A. Schiper, F. Cappello, T. Martsinkevich, A. Guermouche, and T. 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.

[25] C. Wang, F. Mueller, C. Engelmann, and S. 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.

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