

Low-Overhead Fault-Tolerance Support Using DISC Programming Model

Mehmet Can Kurt^{1(✉)}, Bin Ren², and Gagan Agrawal¹

¹ Department of Computer Science and Engineering,
The Ohio State University, Columbus, OH, USA
{kurt, agrawal}@cse.ohio-state.edu

² Pacific Northwest National Laboratory, Richland, WA, USA
bin.ren@pnnl.gov

Abstract. DISC is a newly proposed parallel programming paradigm that models many classes of iterative scientific applications through specification of a domain and interactions among domain elements. Accompanied with an associated runtime, it hides the details of inter-process communication and work partitioning (including partitioning in the presence of heterogeneous processing elements) from the programmers. In this paper, we show how these abstractions, particularly the concepts of *compute-function* and *computation-space* objects, can be also used to leverage low-overhead fault-tolerance support. While *computation-space* objects enable automated application level checkpointing, replicated execution of *compute-functions* helps detect soft errors with low overheads. Experimental results show the effectiveness of the proposed solutions.

1 Introduction

High performance computing is undergoing a significant transformation in the sense that resilience is becoming as equally important as performance. Computing power is constantly being increased with more number of cores, hence with more parallelism. This trend results in a significant decrease in Mean Time To Failure (MTTF) rates in HPC systems due to the large number of components. At the same time, parallel machines are becoming more memory and I/O bound. These two trends together are implying that resilience is not only a major problem, but also the commonly used solutions for fault-tolerance, mostly based on system-level checkpointing, are becoming too expensive. The total cost of fault-tolerance support with checkpointing, which is the sum of the costs of taking checkpoints (which increases as checkpointing frequency increases), the net time spent on recomputation (which increases as checkpointing frequency decreases), and the time spent on restart after a failure, can dominate the actual execution time. An analysis of a 100,000 core job, where each node has a MTTF of 5 years, indicates that these three costs can add up to 65 % of the total execution time, i.e. only 35 % of the time will be productively used [10]. Technology trends indicate that this situation will only get worse in the near future in the sense that MTTF

values will become so small that the time required to complete a checkpoint can exceed the MTTF making the existing approach completely inapplicable [4].

Moreover, in recent years, there is a growing concern about a new class of failures, namely, soft errors. These errors involve bit flips in either processing cores, the memory, or the disk. Although radiation has been considered the main cause of such random bit flips [20], use of smaller and smaller transistors and efforts to improve power-efficiency in hardware are now attributed as the cause of these faults occurring more frequently [25]. Many recent publications have summarized the observed frequency of these faults [10], for example, double bit flips (which cannot be corrected by Error Correcting Codes) occur daily at a national lab's Cray XT5, and similarly, such errors were frequent in BG/L's unprotected L1 cache. Although the traditional solutions to deal with soft errors have been implemented at the hardware level, clearly there is a need for software solutions to this problem.

These developments are imposing new challenges for application programmers. On one hand, they need to be able to manually implement efficient application-level checkpointing and recovery. Even more challenging for them is to implement techniques for dealing with soft errors. One pressing question is whether programming models can help automate fault-tolerant solutions.

In this paper, we address this question in the context of the DISC programming model recently developed by the authors. DISC [15] is a programming model and associated runtime system based on *domain* and *domain element* interaction concepts and particularly targets iterative scientific applications with structured grids, unstructured grids and N-body simulation patterns. While these applications have different communication patterns, they are similar in an important way, i.e., they have an underlying domain, and most of the computation occurs due to the interactions among domain elements. Our programming model supports an API by which the domain, interaction among domain elements, and functions for updating any attributes of these domain elements can be explicitly specified. Starting from this model, inter-process partitioning of the work and the communication is handled automatically by the runtime system. Our previous work has shown how the system is almost as efficient as MPI for homogeneous clusters, while allowing repartitioning of work for dealing with heterogeneous configurations.

In this paper, we examine another important application of this programming model. We extended DISC model so that it also leverages low-overhead fault-tolerance support. We show that the abstractions that DISC model provides to hide the details of process communication and work partitioning/re-partitioning help also identify the main execution state and the functions that are the most susceptible to soft errors. Exposure of such vital program state and instructions is utilized in order to implement two fault-tolerance mechanisms within the runtime. First, with the concept of *computation-space* objects, DISC API makes it feasible to support automated, yet efficient, application-level checkpointing. This as a result can reduce checkpointing overheads significantly. Second, with the concept of *compute-functions*, DISC runtime is capable of detecting soft errors using a partial replication strategy. Here, only the set of instructions most

likely to corrupt the main execution state is executed with redundancy and the results are compared efficiently with computed checksums.

To show the effectiveness of our approach, we have developed two stencil computations, one unstructured grid based computation, and a molecular dynamics mini-application (MiniMD, a representative of a full-scale molecular dynamics application). We first compare the cost of checkpointing in our model, against system-level checkpointing in MPI (which is the only automated solution available today). Next, we compare the performance of DISC implementations with replication support to normal execution without any redundancy and show how further improvements in replication overheads can be achieved.

2 Related Work

Fault-tolerance for high performance computing against hard errors has been extensively studied. Much of this research specifically targets MPI [1, 3, 6, 12, 14, 17, 26]. Recent efforts on optimizing the process include combination of coordinated and uncoordinated checkpointing [23] and compression for reducing the overheads of checkpointing [13]. Another approach is algorithm-level fault-tolerance [2, 5, 7, 19], where properties of an algorithm are exploited (typically to build-in redundancy). While this approach can overcome many of the overheads of general checkpointing, it has two key limitations: (1) as the name suggests, the solution is very specific to a particular algorithm, and (2) the fault-tolerant algorithm needs to be implemented by the programmer manually while developing the application. As for soft errors, the general detection approach is through redundant execution. This redundancy can be achieved at various levels. For instance, in [18], each computing node in execution is paired with a buddy node that performs the same work. Paired nodes checkpoint and exchange their local state periodically and the resulting computations in paired nodes are cross compared through their respective checkpoints. [10] provides a new MPI implementation that creates replica MPI tasks and performs online verification during communication only on MPI messages. Studies in [22, 24] execute all dynamic instructions in a program twice by redundant threads and compare the first and second result. If there is a mismatch, both threads restart execution from the faulty instruction. There have been some efforts to reduce the overheads associated with redundancy; [27] exploits high-level program information at compile time to minimize data communication between redundant threads, whereas [21] explores the partial redundant threading spectrum, in which only a dynamic subset of instructions is duplicated to near single threaded execution performance at the expense of limited fault coverage. [9] combines redundant threading with symptom-based detectors by quantifying the likelihood that a soft-error impacting an instruction creates a symptom such as branch mispredicts or cache misses. Resultingly, it only duplicates the instructions that can not generate any such symptoms. Although the proposed solutions achieve significant reductions in associated overheads, none of them attempts to implement redundancy at the programming model level. As we show in next sections, proper abstractions at

programming model level can expose the most vital program state and instructions and can help automate redundant execution with small overheads.

3 DISC Programming Model

In this section, we present the key concepts of DISC programming model as a background for next section which explains how its abstractions leverage low-overhead fault-tolerance support.

3.1 Domain and Subdomain

DISC model treats the entire input space of an application as a multidimensional *domain*, which consists of *domain elements*. At the beginning of execution, programmers provide information about the domain. This information is used to initialize the runtime system and it includes (1) whether the domain represents a structured grid, an unstructured grid or a particle set, (2) number of dimensions and boundary values for each dimension and (3) the type of interaction among domain elements. Once this information is passed to the runtime, it decomposes the entire domain into non-overlapping rectilinear regions referred as *subdomains* and assigns each subdomain to a processing unit. Since subdomain decomposition and assignment is performed by the runtime, it is able to hold a high-level view of the domain.

As a concrete example, consider a molecular dynamics application such as MiniMD which simulates the motion of a large number of atoms in three-dimensional space throughout a predefined number of time-steps. When implemented using DISC model, the three-dimensional space is treated as an N-body simulation domain and each atom in the simulation corresponds to a domain element. DISC runtime for MiniMD is initialized with the following lines of code;

```
// provide domain information and initialize DISC runtime
DomainProps props;
props.set_ndims(3); // number of dimensions
props.set_min_bounds(0, 0, 0); // x, y, z min-bounds
props.set_max_bounds(XMAX, YMAX, ZMAX); // x, y, z max-bounds
NBodyDomain domain(props);
```

3.2 Attributes

Each domain element in a DISC domain has associated coordinate values. In some domain types such as structured grids, coordinate values of domain elements might stay fixed during the entire execution and can be inferred directly from the boundary values of assigned subdomains. However, for other domain types, they might be updated periodically and their initial values should be explicitly provided by programmers. In addition to coordinates, each domain element can also be associated with a set of *attributes*. For instance, each atom

in MiniMD has three additional attributes that store velocity values of the corresponding atom on x, y and z axis. The key advantage of DISC model is its ability to perform data exchange operations based on the interaction pattern automatically and to re-partition the domain on the fly in presence of heterogeneity by migrating domain elements within the domain. To fulfill both of these promises, programmers register coordinates and attributes of domain elements within each subdomain via DISC API, so that the runtime is informed of the data structures that maintain associated information on each domain element. Using the same example, the code snippet below shows how attributes of domain elements in MiniMD are passed to the runtime through DISC objects called *DoubleAttribute*;

```
DoubleAttribute velocities[3]; // x, y, z velocities
/* fill in attribute object velocities with initial values of x, y, z velocities */
domain.register_attributes(&velocities);
```

3.3 Compute-Function and Computation-Space

In DISC model, each processing unit performs computations for the assigned portion of the domain. In other words, the domain elements that a processing unit processes lie within the boundaries of the subdomain that has been assigned to it by the runtime. DISC requires programmers to express underlying computation, which typically comprises of calculating new values for attributes associated with domain elements, in a single or a set of functions referred as *compute-functions*. Compute-functions generally host the portion of code on which most of the execution time is spent. Programmers specify these functions by passing function pointers to the runtime. At each iteration during a program's execution, the runtime invokes these functions in the order that they are specified.

For each compute-function, programmers explicitly declare one or more objects called *computation-space*. A computation-space object coupled with a compute-function stores the results of computation carried out by that function. It generally contains an entry for each domain element in the corresponding subdomain and programmers perform any updates related to the domain elements directly on the computation-space object itself. This way, the runtime is aware of what additional data structures along with coordinates and attributes describe the domain elements in a subdomain completely. This abstraction leverages automated migration of domain elements within the domain if needed. Note that mapping a value in computation-space to the corresponding domain element can be inferred from domain type in most cases. Otherwise, programmers can pass additional functions to the runtime that dictate this mapping.

In MiniMD, atoms interact with other atoms in a given radius and this interaction results in recomputation of coordinates and velocities of each atom at every time-step. The code snippet below reflects this by defining six computation-space objects (three for new coordinates and three for new velocities). These objects are coupled with the compute-function *minimd_kernel* and passed to the runtime via DISC API;

```
DoubleAttribute computation_space[6]; // new x, y, z coords and velocities
domain.add_compute_function(minimd_kernel, &computation_space);
```

3.4 Interaction Between Domain Elements

As indicated before, a key advantage of DISC model is that the runtime handles communication automatically based on the type of interaction between domain elements. Currently, DISC model supports three types of communication; based on nearest neighbor interactions in stencil kernels, based on radius-based interactions in molecular dynamics applications and based on a list provided explicitly by programmers that dictates pair-wise interactions. Further details for runtime communication generation can be found in [15].

4 Fault-Tolerance Support

We now describe two fault-tolerance approaches that have been implemented for the applications developed using DISC model.

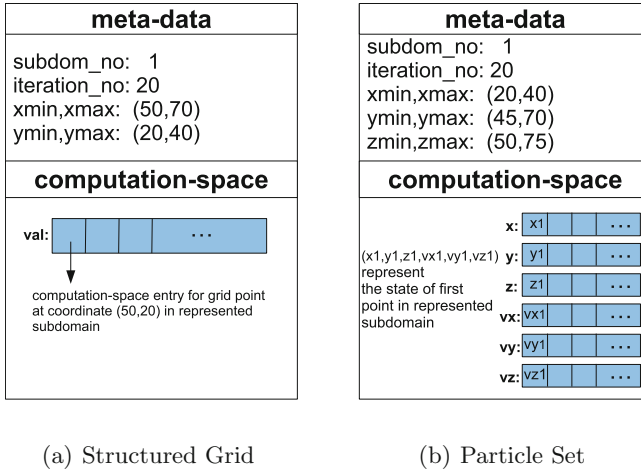


Fig. 1. Sample checkpoint files for a 2D stencil (a) and 3D molecular dynamics application (b). Both files consist of two parts as meta-data and computation-space objects.

4.1 Checkpointing

DISC model automates application-level checkpointing, alleviating the need for expensive system-level checkpointing that is normally used for programming models like MPI. Like any checkpointing-based approach, we assume the existence of a persistent storage where the checkpoint files can be written into.

Two important questions for application-level checkpointing are: (1) when should checkpoints be taken, and (2) what data structures will be needed to restart the computation in case of a failure, and therefore, need to be checkpointed. It turns out that the DISC model simplifies these decisions. Particularly, the end of an iteration of the time-step loop (after data exchange and main computation have been completed by the runtime system) is a natural point for taking the checkpoint. Compared to system-level checkpointing, we get a coordinated checkpoint (in the sense that there is no need for message logging for recovery), while not requiring any time-consuming coordination between processes.

Now, let us return to the question of which data structures need to be checkpointed. DISC model encapsulates the computational progress made on each domain element in objects that we introduced in previous section; *attribute* and *computation-space* objects. At each iteration, attribute objects store the current information associated with domain elements, whereas computation-space objects capture the updates on them performed through compute-functions. As a concrete example, if we consider MiniMD, after each time-step, the attributes and the computation-space objects contain previous and updated coordinate and velocity values of each atom. The collection of attribute and computation-space objects represent the main execution state of applications at any given time. This collection along with the high-level information such as initial domain decomposition (boundaries of each subdomain) can be used to recover the state of DISC runtime and the underlying application completely.

If an application has multiple compute-functions, not all computation-space objects may be live at the end of an iteration of the time-step loop, i.e., certain computation-space objects could have been consumed already. Moreover, some of the attribute objects might entirely depend on and be calculated from a small set of remaining attributes without incurring a significant recomputation cost. This implies that during failure recovery not all of the attributes and computation-space objects are needed to recreate the execution state of domain elements. Some of them can be ignored by the checkpointing mechanism to save bandwidth, hence time, and also storage space. While compiler analysis can provide this information, our model currently asks the programmers to explicitly annotate this information by passing additional arguments during instantiation of these objects. This way, programmers can explore the tradeoff space in checkpointing the entire domain state vs. recalculation of a small portion from saved data structures. Note that any other application state besides the ones associated directly with domain elements should be explicitly checkpointed by programmers. However, considering the computation patterns that DISC model targets, such additional state is limited and recomputed efficiently from checkpointed attribute and computation-space objects.

Checkpointing frequency as well as other important information like the file path where the checkpoint files will reside can be set via DISC API. We insert some meta-data information to the head of checkpoint files including the current iteration number, and also the boundaries of the subdomain that attribute and

computation-space objects represent. This meta-data is utilized to reconstruct the application state during recovery. Figures 1(a) and (b) illustrate the content of sample checkpoint files, which are taken at the 20th iteration of a 2D stencil grid computation and a 3D molecular dynamics application. In both (a) and (b), only the computation-space objects are saved.

Recovery. During recovery from a failure, DISC model is able to restart the computation both with the same or a fewer number of processes, unlike the current checkpointing approaches in MPI, which can only allow restart with the same number of processes. For instance, assuming that there are N processing units in the system before the failure, if the computation is restarted with a fewer number of nodes, say $N - 1$, the domain is decomposed into $N - 1$ subdomains.

Whether with the same or fewer number of nodes, the most critical operation for recovery is to recreate the computational state of a subdomain from existing checkpoint files. If a processing unit has been assigned the same subdomain as before, it will be sufficient to access that subdomain's checkpoint file and load its content into computation-space object in entirety. However, after decomposition, a change in subdomain boundaries is very likely. Therefore, each processing unit may need to read several checkpoint files. In such cases, the metadata information mentioned previously is utilized to filter down the checkpoint files either completely or at least partially, i.e. we check if there is an intersection between processing unit's newly assigned subdomain and the boundaries of the subdomain that the checkpointed computation-space object represents.

Once computation-space objects for the new domain have been reconstructed from the checkpoint files, application can restart from the iteration in which the last checkpoint was taken.

4.2 Replication

Soft error detection has drawn significant attention from community in recent years. Such error detection could be from a variety of sources including hardware or software error detection codes such as ECC, symptom-based error detectors [11] and application-level assertions. One approach to detect such errors is to create two or more independent threads of execution and compare the execution state of different threads. This work has been done at multiple levels – replication at process level [10] or replication at the instruction level [9]. However, trivial replication of the entire program execution and comparison of resulting computation might incur significant overheads. We claim that concepts of *compute-functions* and *computation-space* objects in DISC model can be used to implement a partial replication strategy to reduce associated overheads substantially.

As emphasized before, compute-functions contain the lines of code to which majority of program execution time is devoted. A soft error in combinatorial logic components including register values, ALUs and pipeline latches is most likely to occur when processing cores carry out the instructions expressed in

compute-functions. Since computations, and hence updates on domain elements, defined in compute-functions are directly reflected on the computation-space objects coupled with them, a soft error occurring during the execution of these functions eventually corrupts the computation-space objects, either directly and transitively. This observation suggests that soft errors can be efficiently detected by replication of compute-functions only and cross-comparison of their associated computation-space objects after each iteration. Note that replication mechanism described next assumes that processor components other than the memory are susceptible to soft errors. A produced value is assumed to be resilient once it leaves the processor and is stored back in memory. Control flow variables and memory references are protected by other means such as invariant assertions against the possibility of causing fatal errors such as segmentation faults. Hence, we mainly protect execution against soft errors on calculated values that are used to update computation-space objects.

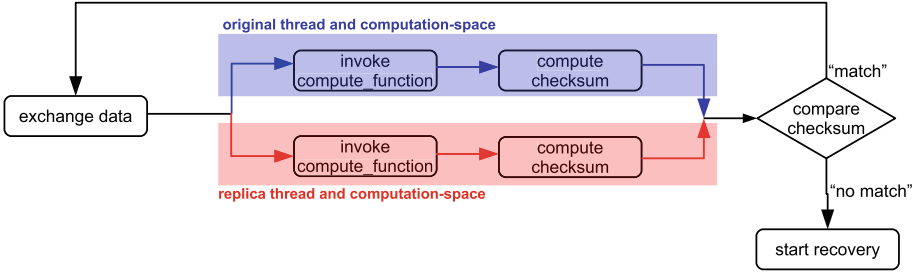


Fig. 2. Flow of execution at each iteration when replication strategy is in use.

Replication Mechanism. Figure 2 demonstrates the execution flow at each iteration when this partial replication strategy is implemented in the DISC runtime. After data exchange operations are performed, the runtime splits the main execution thread into two as *original* and *replica*. Each thread is associated with its own computation-space object, but they both invoke the same compute-function in parallel. During compute-function execution, both original and replica threads use the same set of input space, i.e. attributes of domain elements and any global data structures in application code. Sharing the same memory space, except the computation-space objects, leads to a significant reduction in overall memory footprint of replication strategy.

Currently, the replication strategy in DISC model makes the assumption that compute-functions provided by the programmer are side-effect free, meaning that they do not modify any global data structures. This is mainly to avoid possible race conditions. Note that one can synchronize original and replica threads by pragma directives with respect to the threading library used by the DISC runtime.

Checksum Calculation. After both threads finish executing the compute-function, they calculate a checksum value over their own computation-space

object. We employ integer module operation as the checksum function. Regardless of their data type, we treat the bit representation of values in computation-space objects as an integer and accumulate them into a single sum [16]. After checksum calculation, the two threads merge and checksum values are compared by the main thread. If the values match, application advances to the next iteration. Otherwise, DISC runtime ceases the execution and informs the programmer that a soft error has been detected and a recovery procedure should be initiated.

Improvements for Cache Utilization. The initial replication scheme calculates checksums over computation-space objects once individual threads finish execution of compute-functions. Although checksum calculation can be performed quite efficiently, especially in architectures with vector units, accessing the entire computation-space objects once again leads to a large number of cache misses, and hence to high overheads, especially when computation-space objects are large. To remedy this, we present an improvement on top of the plain replication scheme presented previously. Instead of performing it in a separate step, we incorporate checksum calculation directly into compute-functions. Particularly, pure compute-functions provided by programmers are modified in a way that entries in a computation-space object contribute to the checksum on the fly, right after they are assigned a value. *On the fly checksum calculation* increases temporal locality of overall replication strategy and helps us avoid the data access costs incurred by an isolated checksum calculation phase.

Another source of overhead is the need to create a second copy of computation-space objects. Having additional computation-space objects for replica threads both increases the total memory footprint and at the same time diminishes overall cache utilization. Thus, as a second improvement, we avoid creating replica computation-space objects by modifying compute-functions further. Particularly, assignments to computation-space objects in replica thread are replaced by instructions that accumulate the assigned variables to the checksum values instead. Having *no replica computation-space object* in replica threads results in further improvements in data locality. In the next section, we demonstrate how these two optimizations affect performance of replication strategy, especially for applications with large output space.

5 Experiments

In this section, we present results from a number of experiments we conducted to evaluate the fault-tolerance solutions that we implemented within DISC model. Our evaluation is based on four applications. We chose one molecular dynamics application (MiniMD), one application involving an unstructured grid (Euler), and two smaller kernels involving stencil computations (Jacobi and Sobel).

5.1 Checkpointing

One of the key advantages of DISC model is the support for automated application-level checkpointing. We now show how the cost of checkpointing with our approach

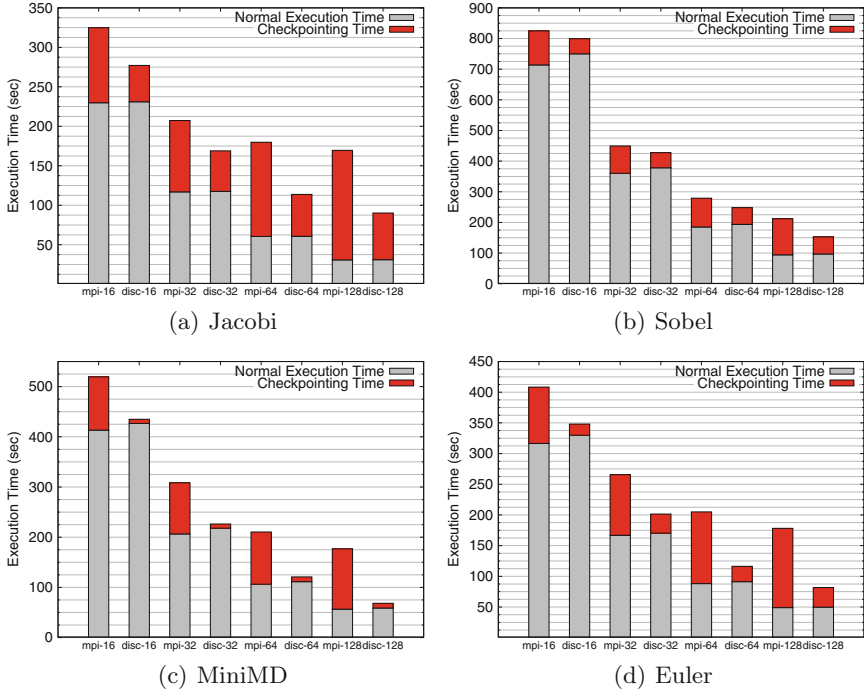


Fig. 3. Normal execution and checkpointing times of MPI and DISC implementations of four applications with varying number of nodes.

compares with the only automated solution currently available with MPI, which is system-level checkpointing. Moreover, we also examine how the total execution time of our system and MPI versions compare, when checkpointing overheads are included.

For checkpointing support in MPI implementations, we used MPICH2-BLCR, which is one of the most popular system-level checkpoint/restart libraries. MPI versions of all evaluated applications have been written by ourselves, except MiniMD which was obtained from the Mantevo suite¹. Experiments in this section are performed on a cluster where each node has two quad-core 2.53 GHz Intel(R) Xeon(R) processors, with 12 GB RAM, executing RedHat Enterprise Linux Server release 6.1, and Gigabit ethernet as the interconnect. Our programming model is implemented in C++ language and uses MPICH2 (version 1.4.1p1) as the underlying communication library. The comparisons have been performed over a varying number of nodes ranging between 16 and 128 (with only one core at each node), consistent with our focus on distributed memory parallelism. Both in this and next section, we repeated each experiment 5 times and report the average results.

¹ Please see <https://software.sandia.gov/mantevo>.

Figure 3(a) and (b) demonstrate the execution times of Jacobi and Sobel, as we increase the number of nodes. Gray portions of the bars correspond to normal execution times, whereas red portion on top of each bar shows the additional time spent for checkpointing. For both applications, we use a grid structure with 400 million elements, execute them for 1000 iterations and trigger checkpoint mechanism every 250 iterations. Compared to the MPI versions, our model’s implementations have average overheads less than 1 % for Jacobi and 4 % for Sobel in normal execution times. The size of each global checkpoint in Jacobi and Sobel is 6 GB for MPI and 3 GB for our model. Corresponding figures show that checkpointing operations in our model are completed approximately in half of the time than MPI.

Figure 3(c) and (d) report the same results for MiniMD and Euler. In MiniMD, we simulate the behavior of 4 million atoms, whereas we use 12 million nodes for Euler. We run each application for up to 1000 iterations and take checkpoints every 100 iterations. Results show that implementing MiniMD and Euler with DISC brings an average overhead less than 5 % in normal execution without checkpointing. In MiniMD, each global checkpoint of MPI version is nearly 2 GB in size, whereas with our programming model, the application-level checkpoint is only 192 MB. Consequently, on the average, checkpointing time of MPI is nearly 12 times higher. As the number of nodes increases, checkpointing times increase, due to the fact that more nodes are contending for pushing the data to file system at the same time. In Euler, the global snapshot size is again 2 GB for MPI, and 640 MB with our programming model. As a result, the time required for checkpointing in MPI is nearly 4 times higher.

It is also useful to note that in all cases, after adding the normal execution and checkpointing times, our model is faster. In some of the cases, particularly, execution of MiniMD and Euler on 128 nodes, our model reduces the total execution time at least by a factor of 2, when checkpointing overheads are included. Furthermore, we can see that with increasing number of nodes, as well as with increasing complexity of applications, the relative advantage of our model increases. The former is because of increasing contention for I/O related to checkpointing, whereas, the latter is because a full application has many more structures than those that need to be checkpointed at the application level. Because Jacobi and Sobel are small templates, the application-level checkpoint is nearly 50 % of the size of system-level checkpoint. In comparison, for a more complex application like MiniMD, the ratio is close to 10 %. Thus, we can see that for most applications, we can expect significant performance from our model.

5.2 Replication

Next, we present the results for DISC implementations of the previous applications, when we replicate compute-function execution in each process. We evaluate our partial replication approach on Intel Xeon Phi 7110P many-core coprocessor. The reason for choosing this architecture is that many-core systems are likely to

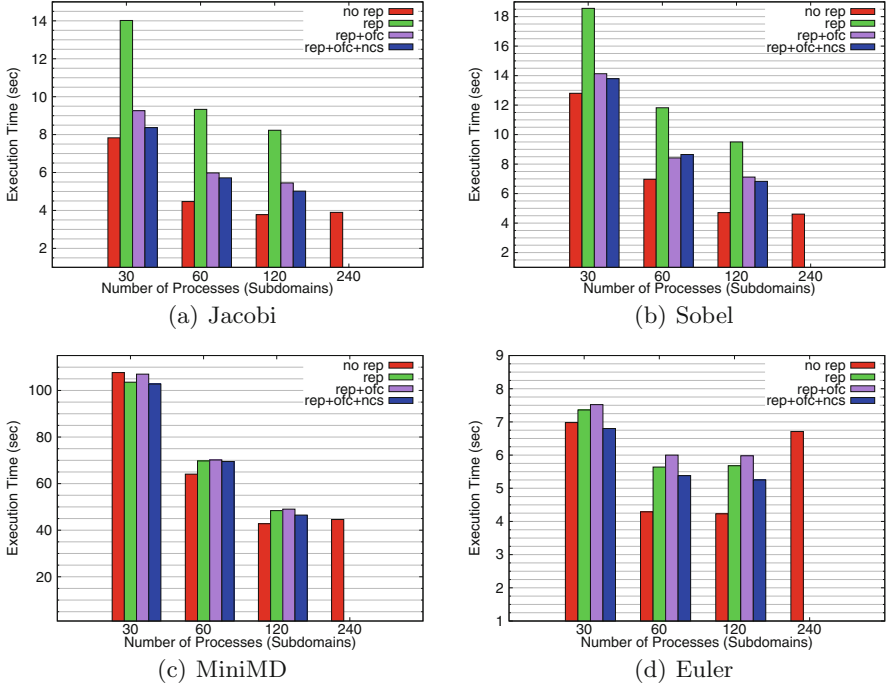


Fig. 4. Execution times of four applications without any replication (*no rep*), with plain replication (*rep*) and replication with improvements for cache utilization (*rep+ofc* and *rep+ofc+ncs*). Execution times for *no rep* with 1 process are 307.9, 398.9, 2686.2 and 213.2s in Jacobi, Sobel, MiniMD and Euler, respectively. The same execution times for the best replication version *rep+ofc+ncs* are 316.2, 474.5, 2738.3 and 214.8s.

be common in the exascale era, where soft errors will also be more likely. Specifically, the coprocessor we have used has 61 cores running at 1.1 GHz with 32KB L1 cache, 512KB L2 cache per core and 8GB device memory for all cores, and is capable of running 244 hardware threads with hyperthreading support. All applications were compiled by Intel icpc-13.1.0 compiler with -O3 optimization with auto vectorization flag on. Each process replicates the compute-function execution step using OpenMP multi-threading library. We run all applications for 100 iterations. To mitigate the impact of system noise, we dedicate core0 of Xeon Phi to the OS and pin DISC processes to hardware threads between core1 and core60. Original and replica threads in each process are pinned to the same core, except configurations where we have 1 and 30 processes.

Figure 4(a) and (b) present the replication results for Jacobi and Sobel. For each application, we compare the performance of four DISC versions; (1) execution without any replication (*no rep*), (2) execution with plain replication (*rep*), (3) execution with replication and on the fly checksum calculation (*rep+ofc*), and finally (4) execution with replication, on the fly checksum calculation and no replica computation-space (*rep+ofc+ncs*). All DISC versions are run with 1,

30, 60, 120 processes. For 240 processes, we only report the results for no replication version, since the replication versions utilize all of the 240 hardware threads with 120 processes. The figure shows that for Jacobi at 120 processes DISC replication versions *rep*, *rep+ofc* and *rep+ofc+ncs* have 118 %, 44 % and 33 % overheads, respectively, over execution with no replication. Note that because the 240 thread *no rep* version does not have better performance over the 120 thread version, the results from 120 threads can be used to establish overheads of replication over the most efficient execution without replication. For Sobel, with the same number of processes, the overheads are 102 %, 51 % and 45 %. These results indicate that two improvements over the plain replication scheme lead to significant reductions in total overhead by reducing data access costs during checksum calculation and improving overall cache utilization.

Figure 4(c) and (d) present the results for MiniMD and Euler. At 120 processes, DISC replication strategy causes 13 %, 15 % and 9 % overheads in MiniMD, respectively for *rep*, *rep+ofc* and *rep+ofc+ncs* versions. In Euler, the same overheads are 34 %, 41 % and 24 %. Although the overheads with the plain replication version itself is quite small, we see that the suggested improvements do not lead to substantial benefits compared to Jacobi and Sobel. This is mainly due to the fact that computation-space objects in MiniMD and Euler have a smaller size and they fit in the L2 cache of Xeon Phi cores. Another potential reason is the following. Xeon Phi employs software and hardware-based data prefetching to reduce data access latencies. The prefetching mechanism works very aggressively for stencil kernels and accessing the same data within a core both by original and replica threads might lead to capacity and conflict misses. Furthermore, an existing analysis on Xeon Phi in [8] reports drops in bandwidth when different threads access the same memory space simultaneously due to the effects of contention at the interconnect level. Hence, we believe that any data locality optimization such as *on the fly checksum calculation* and *no replica computation-space object* for kernels such as stencils result in substantial improvements. On the other hand, due to the irregular data access patterns in MiniMD and Euler, the amount of data prefetching is limited. The overhead for plain replication is not too high to begin with and the improvements in *rep+ofc* and *rep+ofc+ncs* versions are less visible.

Table 1. Error detection rates for plain replication (*rep*) and replication with on the fly checksum and no replica computation-space object (*rep+ofc+ncs*) versions both without and with soft error injection.

	Normal execution		With error injection	
	rep	rep+ofc+ncs	rep	rep+ofc+ncs
Jacobi	0 %	0 %	100 %	100 %
Sobel	0 %	0 %	100 %	100 %
MiniMD	0 %	0 %	100 %	100 %
Euler	0 %	0 %	24 %	100 %

As the last experiment, we show how effective DISC partial replication strategy is in detecting soft errors. Table 1 reports error detection rates when the four applications are run both when there is no soft error occurrence during execution and when a single soft error is injected. Error injection is done manually by flipping a single bit of a random stack variable during the execution of compute-functions. We repeat the same experimental setup for two versions; plain replication (*rep*) and replication with on the fly checksum and no replica computation-space (*rep+ofc+ncs*). Each configuration is performed 50 times and error detection rates show how many times DISC detected an error in these runs as a percentage. Results show that when there is no soft error injection, error detection rate for both versions is 0 % meaning that DISC replication strategy does not produce any false positives. Moreover, in Jacobi, Sobel and MiniMD, both versions are able to detect injected soft errors and achieve 100 % error detection rate. As the only exception, in Euler, plain replication version detects only 24 % of injected errors, whereas *rep+ofc+ncs* again achieves a 100 % detection rate. This is due to the fact that in Euler each corrupted stack variable makes two contributions to the computation-space objects, one being positive and the other negative. When checksums are calculated in plain replication scheme, positive and negative contributions seem to cancel out each other reducing overall detection rate. In contrary, *rep+ofc+ncs* version is insusceptible to such cancellation, since checksums are calculated by using the corrupted assigned values directly and ignoring their sign.

6 Conclusion

In this paper, we presented how DISC, a parallel programming model for iterative scientific applications based on structured, unstructured grids and N-body simulations, is extended to leverage low-overhead fault-tolerance support. We showed that the existing abstractions in DISC model for automated inter-process communication and work partitioning/re-partitioning can be also used for automated application-level checkpointing and replicated execution to detect soft error occurrences. The experimental evaluation shows that checkpointing in DISC model provides significant improvements over system-level checkpointing scheme and soft errors can be detected by a partial replication strategy with low overheads.

Acknowledgments. This work was supported by National Science Foundation under the award CCF-1319420, and by the Department of Energy, Office of Science, under award DE-SC0014135.

References

1. Agbaria, A., Friedman, R.: Starfish: fault-tolerant dynamic MPI programs on clusters of workstations. In: 1999 Proceedings of the Eighth International Symposium on High Performance Distributed Computing, pp. 167–176 (1999)

2. Arnold, D., Miller, B.: Scalable failure recovery for high-performance data aggregation. In: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), pp. 1–11, April 2010
3. Bouteiller, A., Cappello, F., Herault, T., Krawezik, G., Lemarinier, P., Magniette, F.: MPICH-V project: a multiprotocol automatic fault tolerant MPI. *Int. J. High Perform. Comput. Appl.* **20**(3), 319–333 (2006)
4. Cappello, F.: Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *Int. J. High Perform. Comput. Appl.* **23**(3), 212–226 (2009)
5. Chen, Z.: Algorithm-based recovery for iterative methods without checkpointing. In: Proceedings of the 20th International Symposium on High Performance Distributed Computing, HPDC 2011, pp. 73–84. ACM, New York (2011)
6. Coti, C., Herault, T., Lemarinier, P., Pilard, L., Rezmerita, A., Rodriguez, E., Cappello, F.: Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI. In: Proceedings of the ACM/IEEE Conference on Supercomputing, SC 2006. ACM, New York (2006)
7. Davies, T., Karlsson, C., Liu, H., Ding, C., Chen, Z.: High performance linpack benchmark: A fault tolerant implementation without checkpointing. In: Proceedings of the International Conference on Supercomputing, ICS 2011, pp. 162–171. ACM, New York (2011)
8. Fang, J., Varbanescu, A.L., Sips, H., Zhang, L., Che, Y., Xu, C.: An Empirical Study of Intel Xeon Phi. ArXiv e-prints, October 2013
9. Feng, S., Gupta, S., Ansari, A., Mahlke, S.: Shoestring: Probabilistic soft error reliability on the cheap. *SIGPLAN Not.* **45**(3), 385–396 (2010)
10. Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., Brightwell, R.: Detection and correction of silent data corruption for large-scale high-performance computing. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 78:1–78:12. IEEE Computer Society Press, Los Alamitos (2012)
11. Hari, S.K.S., Adve, S.V., Naeimi, H.: Low-cost program-level detectors for reducing silent data corruptions. In: DSN, pp. 1–12 (2012)
12. Hursey, J., Squyres, J., Mattox, T., Lumsdaine, A.: The design and implementation of checkpoint/restart process fault tolerance for open MPI. In: IEEE International Parallel and Distributed Processing Symposium, IPDPS 2007, pp. 1–8, March 2007
13. Islam, T.Z., Mohror, K., Bagchi, S., Moody, A., de Supinski, B.R., Eigenmann, R.: Mcengine: A scalable checkpointing system using data-aware aggregation and compression. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 17:1–17:11. IEEE Computer Society Press, Los Alamitos (2012)
14. Kranzlmüller, D., Kacsuk, P., Dongarra, J.: Recent advances in parallel virtual machine and message passing interface. *Int. J. High Perform. Comput. Appl.* **19**(2), 99–101 (2005)
15. Kurt, M.C., Agrawal, G.: Disc: A domain-interaction based programming model with support for heterogeneous execution. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, pp. 869–880. IEEE Press, Piscataway (2014)
16. Maxino, T., Koopman, P.: The effectiveness of checksums for embedded control networks. *IEEE Trans. Dependable Secure Comput.* **6**(1), 59–72 (2009)

17. Moody, A., Bronevetsky, G., Mohror, K., de Supinski, B.R.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010, pp. 1–11. IEEE Computer Society, Washington, DC (2010)
18. Ni, X., Meneses, E., Jain, N., Kalé, L.V.: ACR: Automatic checkpoint/restart for soft and hard error protection. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2013, pp. 7:1–7:12. ACM, New York (2013)
19. Plank, J., Kim, Y., Dongarra, J.: Algorithm-based diskless checkpointing for fault tolerant matrix operations. In: Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, FTCS-25. Digest of Papers, pp. 351–360, June 1995
20. Quinn, H., Graham, P.: Terrestrial-based radiation upsets: a cautionary tale. In: 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2005, pp. 193–202, April 2005
21. Reddy, V.K., Rotenberg, E., Parthasarathy, S.: Understanding prediction-based partial redundant threading for low-overhead, high- coverage fault tolerance. *SIGARCH Comput. Archit. News* **34**(5), 83–94 (2006)
22. Reinhardt, S.K., Mukherjee, S.S.: Transient fault detection via simultaneous multithreading. In: Proceedings of the 27th Annual International Symposium on Computer Architecture, ISCA 2000, pp. 25–36. ACM, New York (2000)
23. Riesen, R., Ferreira, K., Da Silva, D., Lemarinier, P., Arnold, D., Bridges, P.G.: Alleviating scalability issues of checkpointing protocols. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 18:1–18:11. IEEE Computer Society Press, Los Alamitos (2012)
24. Rotenberg, E.: AR-SMT: a microarchitectural approach to fault tolerance in microprocessors. In: Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, Digest of Papers, pp. 84–91, June 1999
25. Schroeder, B., Pinheiro, E., Weber, W.-D.: DRAM errors in the wild: A large-scale field study. In: Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2009, pp. 193–204. ACM, New York (2009)
26. Stellner, G.: CoCheck: checkpointing and process migration for MPI. In: Proceedings of the 10th International Parallel Processing Symposium, IPPS 1996, pp. 526–531, April 1996
27. Wang, C., Kim, H.-S., Wu, Y., Ying, V.: Compiler-managed software-based redundant multi-threading for transient fault detection. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO 2007, pp. 244–258. IEEE Computer Society, Washington, DC (2007)

Languages and Compilers for Parallel Computing
28th International Workshop, LCPC 2015, Raleigh, NC,
USA, September 9-11, 2015, Revised Selected Papers
Shen, X.; Mueller, F.; Tuck, J. (Eds.)
2016, X, 319 p. 139 illus. in color., Softcover
ISBN: 978-3-319-29777-4