Automatic Risk-based Selective Redundancy for Fault-tolerant Task-parallel HPC Applications

Omer Subasi
Pacific Northwest National Laboratory
omer.subasi@pnnl.gov

Osman Unsal
Barcelona Supercomputing Center
osman.unsal@bsc.es

Sriram Krishnamoorthy
Pacific Northwest National Laboratory
sriram@pnnl.gov

ABSTRACT

Silent data corruption (SDC) and fail-stop errors are the most hazardous error types in high-performance computing (HPC) systems. In this study, we present an automatic, efficient and lightweight redundancy mechanism to mitigate both error types. We propose partial task-replication and checkpointing for task-parallel HPC applications to mitigate silent and fail-stop errors. To avoid the prohibitive costs of complete replication, we introduce a lightweight selective replication mechanism. Using a fully automatic and transparent heuristics, we identify and selectively replicate only the reliability-critical tasks based on a risk metric. Our approach detects and corrects around 70% of silent errors with only 5% average performance overhead. Additionally, the performance overhead of the heuristic itself is negligible.

CCS CONCEPTS

• **Computer systems organization → Redundancy**;

KEYWORDS

Fault-tolerance, selective redundancy, task-parallelism, dataflow

ACM Reference Format:
Omer Subasi, Osman Unsal, and Sriram Krishnamoorthy. 2017. Automatic Risk-based Selective Redundancy for Fault-tolerant Task-parallel HPC Applications. In *ESPM2'17: ESPM2'17: Third International Workshop on Extreme Scale Programming Models and Middleware, November 12–17, 2017, Denver, CO, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3152041.3152083

1 INTRODUCTION

As we get closer to the exascale time-frame, fault-tolerance is becoming one of the main concerns along with power and energy consumption in High Performance Computing (HPC). The mean time between errors is expected to be on the orders of hours or even minutes [6] for future HPC and exascale systems. Moreover the extent and impact of exascale failure types such as silent data corruptions (SDCs) or silent errors could present a significant threat for large-scale application reliability [10]: Such errors are not detected and applications produce wrong results. Another type of errors that is common in HPC systems is the fail-stop errors: The computing process/unit aborts execution and loses all computation and data. Hardware-only solutions are not expected to handle these two types of errors at exascale error rates [8]. Thus software based fault-tolerance solutions are needed to complement the hardware based ones. Moreover it is imperative to address both error types at the same time which is what happens in reality. However, very few studies address them simultaneously.

Meanwhile task-based and data-driven programming models (PM) are becoming widely used in HPC community. As an example we see the adaptation of task-based dataflow parallelism in OpenMP 4.0 [11] and Intel Threading Blocks [1]. The performance of dataflow parallelism is shown to be higher than that of the traditional fork-join model due to amount of parallelism achieved [2]. Moreover, these models provide unique advantages in terms of fault tolerance: In task-based dataflow programs, the failure containment is better than traditional parallel programs (for example those implemented with Pthreads) since an error is most likely to cause only the failure of the single task in which it occurs rather than the entire application. Furthermore, even if the error propagates to other tasks, having a dataflow model facilitates tracing the error through dependent tasks to its root cause. Additionally, the asynchronous nature of dataflow computation enables low-overhead error recovery by overlapping recovery of faulty task(s) with execution of other tasks that are independent of the faulty task(s). Therefore we find that it is important to provide fault-tolerance support for task-parallel HPC applications using data-driven execution model to mitigate fail-stop errors and SDCs.

The straightforward approach involves complete task redundancy, that is, to replicate and checkpoint all application tasks. In one possible implementation, tasks are duplicated and tasks' inputs that are available through dataflow information are checkpointed at the beginning of their computations. When both replicas finish, replica outputs are compared for SDC detection and in case of a mismatch, a third instance of the task run is performed after restoring the inputs for a majority vote. Our results indicate that complete task redundancy is scalable even for high error rates and the average fault-free performance overhead is 3%. However the complete task redundancy is only needed for very high error coverage. In addition, although asynchronous execution and data-awareness help to limit overheads, the 2× resource cost of complete task redundancy may be high and/or not required. Hence in this work, we introduce a partial task redundancy scheme to reduce resource costs while providing sufficient level of fault-tolerance.

We develop an effective and efficient runtime heuristic that selectively replicates the most reliability critical tasks for a reasonable trade-off between performance and error coverage. The heuristic selects tasks based on their risks solely utilizing existing knowledge at runtime. Therefore, no additional bookkeeping is added to the

ESPM2'17, November 12–17, 2017, Denver, CO, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5133-1/17/11…$15.00
https://doi.org/10.1145/3152041.3152083

runtime. In Section 3, we discuss how the risk metric is established through *component analysis*. Our heuristic is automatic and completely transparent to user, application and operating system (OS). Moreover, it is very lightweight having negligible performance overhead. Our motivation is to design a heuristic that is as effective, efficient and low-cost as possible while not requiring to keep any extensive information at runtime or any expensive offline application profiling.

Finally, we compare our heuristic to the programmer insights and to the baseline where tasks are selected for replication randomly. Results indicate that our heuristic is significantly more effective and efficient than random task selection.

Our main contribution is the development of a partial task redundancy framework for task-parallel dataflow HPC applications. To the best of our knowledge, this is the first work that explores the risk-based automatic selective task redundancy for task-parallel data-driven programming. Our main contributions are:

- A runtime heuristic that automatically selects the most reliability critical tasks for partial protection providing around 70% detected and corrected errors with only 5% overhead on average. Moreover our heuristic itself has negligible performance overhead.
- For the runtime heuristic, component analysis for each benchmark to identify the most reliability-critical tasks or task components.
- Detailed evaluation and comparison to other techniques.

Our research findings in this study are:

- Our studies show that the risk-based approach can be very effective yet at the same time very lightweight.
- The comparison to the programmer knowledge based technique [18] and random task selection shows that our heuristic outperforms random task selection and is on par with the programmer knowledge based technique. However, while the programmer knowledge based method requires expert application knowledge, the risk based method does not.
- Our technique addresses the silent and fail-stop errors at the same time and has negligible performance overhead.
- Component analysis shows that the most important components or tasks, in terms of reliability, differ among the applications.

The rest of the paper is organized as follows: Section 2 provides background and our motivation. Section 3 details the design of our proposal. Section 4 discusses the experimental evaluation. Section 5 surveys related work. Finally, Section 6 concludes this study.

2 BACKGROUND AND MOTIVATION

In this section, we provide background on error types and the implementation infrastructure. We then discuss our motivation to leverage task-parallel dataflow programming.

2.1 Error Categorization and Failure Model

Faults manifest themselves as errors which can then cause applications or system failures. Errors can lead to three outcomes. First, they can be masked which means their presence does not lead to any failure, such as corrected memory errors by the memory ECC. Second, some errors lead to application/OS crash or stop which we call fail-stop errors (stop failures). As an example, multi-bit errors not detected the memory system ECC can cause application crash. The third class of errors is silent errors or silent data corruptions (SDC). In SDC, the error is not detected, and the application produces incorrect results. Recent studies such as [15, 17] show that SDCs can pose an important threat to the integrity of the results of applications. Our failure model includes both SDCs and fail-stop errors. By addressing both error types, the most significant errors that are expected to occur in real systems can be detected and corrected.

2.2 The Implementation Infrastructure

We implement our proposal in the publicly available OmpSs [9] task-based PM and its publicly available dataflow runtime Nanos [21]. However, the ideas presented in this study are general and applicable to other task-based data-driven platforms or PMs. Nevertheless the performance of the OmpSs PM and its runtime is shown to be comparable to the implementations of OpenMP [3, 4]. OmpSs has been a motivating infrastructure to investigate and implement dataflow task parallelism in OpenMP 4.0. Moreover, OmpSs introduced the hybrid OmpSs+MPI model which combines dataflow computation with the message passing programming providing performance improvements and is on par with MPI [14].

OmpSs extends OpenMP with new data directionality clauses. These simple and straightforward directionality clauses, i.e., the task inputs and outputs, are provided by the programmer. Its Nanos runtime dynamically generates a task dependency graph based on data directionality and code structure. If a task has an input argument that is an output argument of another task, this would create a dependency between them and the task with the input argument will wait until the execution of the other task is finished. The runtime enqueues a task for execution when all the inputs that the task needs are produced. An enqueued task is executed when a core is available. Note that a task runs when all its inputs are ready; this means that, in classical dataflow fashion, task execution can be out of program order. The dataflow execution enables asynchronous parallelism which is the root for high performance and efficient fault-tolerance. When a task finishes its execution, it releases its dependencies, effectively enabling the tasks waiting on its arguments to start. The reader is advised to consult [9] for details on OmpSs.

2.3 Motivation for Leveraging Task-parallel Dataflow Programming

We now present the inherent advantages of task-parallel dataflow programming which favor fault-tolerance:

Better Failure Containment: Tasks provide better failure containment since they contain failures within their computations rather than the entire application. The restart of the entire application is mostly avoided this way.

Asynchronous Execution and Overlap of Computation and Recovery: The asynchronous nature of dataflow execution enables low-overhead error recovery by overlapping recovery of faulty task(s) with execution of other tasks that are independent of the faulty task(s).

Minimal Checkpoint Size: Task inputs are available through programmer annotations which represent the minimal state of the tasks needed to be checkpointed for the fault recovery. In contrast, solutions in OpenMP typically checkpoint the entire address space of the application since the runtime is unaware of the exact application state to protect.

Easiness of SDC Detection: Since the error propagation is only possible through the task outputs, the detection of SDC can be easily done by the comparison of the outputs. This prevents the need for an additional heavyweight SDC detection mechanism such as the technique proposed by Fiala et al. [10].

Enabler for Heuristics: By the knowledge of the task arguments at runtime, very effective and low-overhead heuristics can be designed for selective task redundancy.

Better Performance: The performance of dataflow parallelism is shown to be higher than the traditional fork-join parallelism due to amount of parallelism achieved [2].

3 DESIGN OF OUR FRAMEWORK

This section first presents our fault injection mechanism in Section 3.1. It then details the baseline and the selective task replication and checkpointing designs in Section 3.2 and Section 3.3 respectively. We use *task redundancy* to refer to task replication and checkpointing together.

3.1 Fault Injection Mechanism

We implement our fault injection mechanism to evaluate task redundancy and to have different injection capabilities to corrupt the task state at runtime to assess our partial task protection schemes. We inject faults in the state of the tasks by flipping bits of the outputs. We support two different injection settings where fault injection i) is based on fixed fault rates per task or ii) is based on the application execution time and input size. Injection is performed at the end of the execution of a task based on the chosen setting. In a chosen setting, if a fault is to be injected to the task then we flip as many bits as specified beforehand. Each flip is a random bit of a random element of the task output. In our experiments fault arrivals follow an exponential distribution for the second setting – in the first fixed fault rate setting, it is not possible to enforce a specific distribution.

3.2 Complete Task Redundancy Design

We implement the baseline (complete) task redundancy mechanism via task duplication and checkpointing of task inputs in the runtime. Figure 1 summarizes the baseline design. When a task is ready to execute, the runtime duplicates the task and issues both tasks for execution in addition to the checkpointing of the inputs of the original to main memory. When it finishes its computation, the original task waits for the duplicate task. After the duplicate also finishes its computation, the outputs are compared[1] and if they differ, the original is reissued after restoring its inputs from the checkpoint. After the reexecution of the original, the majority vote of the three runs is taken as the final output. If the outputs are the same, the duplicate and the checkpoint is removed. In case a duplicate crashes due to a fail-stop error, the runtime ignores it and continues with the other duplicate. If both duplicates crash, then the runtime aborts.

[1]Although typically bitwise comparison is used, other comparison methods such as relative error checking can be easily deployed at runtime.

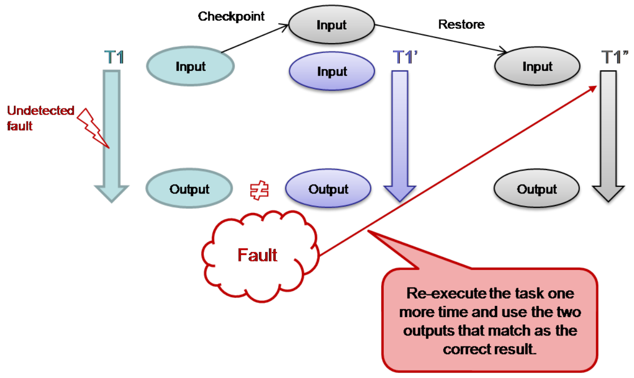
Checkpoint
Input
Restore
T1
Input
Input
T1'
Input
T1''
Undetected fault
Output
Output
Output
Fault
Re-execute the task one more time and use the two outputs that match as the correct result.

Figure 1: Task Redundancy Scheme

Another alternative design is to employ triplication and majority vote for the fault-free case. The advantage of this approach is that since we detect and recover; there is no need for keeping the checkpoints. However, in our experience, this approach is only advisable for very high SDC rates. In any case, we opt for duplex execution for fault-free, and triplex only for fault-recovery since (a) our checkpointing overheads are small and (b) we avoid running a third task for the common case of fault-free execution.

3.3 Selective Task Redundancy Design

The baseline task redundancy design protects, *i.e. duplicates and checkpoints*, all tasks in a program. However in the cases where the protection of all tasks is not needed, partial task protection can provide both required level of fault-tolerance and reduce costs such as power, energy, memory usage. Thus we design and implement a highly lightweight mechanism for selective task protection. In our automated mechanism, we implement a runtime heuristic that automatically selects the tasks to be protected according to our empirically determined criticality measure. It is transparent to the user/programmer since no annotations or modifications to applications are required.

To deploy our technique in a deterministic manner, incorporating a quota ability would be adequate. We leave it as a future work to implement and experiment with this simple quota functionality.

We have developed a runtime heuristic that automatically protects tasks in terms of their criticality defined by a risk function. The risk of a task t is defined to be:

$$risk(t) = (w_i \times ISize(t) + w_o \times OSize(t)) \times w_s \times Succ(t) \quad (1)$$

where $ISize(t)$ is the size of the inputs of t, $OSize(t)$ is the size of the outputs of t, $Succ(t)$ is the number of the immediate successors of t in the task dependency graph. The coefficients or weights w_i, w_o and w_s are determined by *component analysis* which we will elaborate below.

The rationale behind the risk definition is to consider memory usage and task dependency relation. The size of inputs and outputs is a good hint for the memory space used by tasks. Moreover, for parallel applications it has been established that the probability of a fault occurring in a task is associated with the memory space it uses [13]. For this reason, we use task input and output sizes in the risk function. In addition, the risk of a task is positively correlated with the number of its immediate successors as a fault occurring in a task might affect a large number of tasks if it has a large number of immediate successors. We compute the risk of a task without the need for a profiling run since all the necessary information above is available at runtime when a task is ready for execution.

To be able to automatically and dynamically select a task, our mechanism maintains a *global task risk*. Each time a decision is made about whether to protect a task, we update the global task risk as 70 percent of the current global risk value plus 30 percent of the risk of that task. These percentages were determined empirically. We decide updating the global task risk this way to provide some damping when a task with a much higher risk is encountered. Our heuristic computes the task's risk and compares it against the global tasks risk. If the risk of the task is smaller than the global risk then the task is not protected. Otherwise the task is protected. Note that our heuristic only maintains a global variable and performs a simple risk calculation for each task. As a result, it has no significant overhead.

As our running example, Algorithm 1 depicts Sparse LU factorization which is one of our benchmarks which are included in Table 2. It has four phases - *lu0*, *fwd*, *bdiv* and *bmod* - and ported to OmpSs simply by annotating the inputs and outputs of the functions. A is the input matrix and NB is the number of the blocks determining the granularity of tasks. Figure 2 shows the complete color-coded task dependency graph of a Sparse LU computation. The rectangles on the graph show the set of ready tasks to be executed at some particular time. We want to compute the risks of different types of task instances. Let us take four tasks that are instances of *lu0* having 4 immediate successors, *fwd* having 3 immediate successors, *bdiv* having 3 immediate successors, and *bmod* having 1 immediate successor respectively among the ready tasks. Let the block size be BS. Then each task argument's size is BS^2. Let $X = BS^2$. The *lu0* has an input, *fwd* and *bdiv* have an input and an output, and *bmod* has two inputs and an output. In addition as we will see shortly, the weights w_i and w_o are roughly twice as w_s. For simplicity, let $w_i = 2$, $w_o = 2$ and $w_s = 1$. Then according to Equation 1, the risk of the *lu0*, *fwd*, *bdiv* and *bmod* task is $8X$, $12X$, $12X$ and $6X$ respectively. This shows that the instances of *fwd* are *bdiv* more risky/critical than the those of *lu0* and *bmod* at that particular time.

Component Analysis:

To analyze the effect of different task components such as the size of inputs or outputs, the number of inputs, outputs, immediate successors or predecessors of a task, we use a method that we call *component analysis*. We use this method to determine the weights in the risk definition in a systematic manner. We set the risk of a task to be one of these components. We run fault-injected experiments with our fault injection mechanism in which we measure the time overhead and the number of corrupted entries in the final program output for each component. We then define *relative effectiveness (RE)* to evaluate which component is effective in terms of overhead

Algorithm 1: Sparse LU Algorithm

```
Input: $A$: Input matrix to be factorized into LU.
Input: $NB$: The number of blocks tiling the input matrix.
for ($kk = 0; kk < NB; kk++$) do
    #pragma omp task inout(A[kk][kk])
    lu0(A[kk][kk]);
    for ($jj = kk + 1; jj < NB; jj++$) do
        if ($A[kk][jj] != NULL$) then
            #pragma omp task input(A[kk][kk])
            fwd($A[kk][kk]$, $A[kk][jj]$);
        end
    end
    for ($ii = kk + 1; ii < NB; ii++$) do
        if ($A[ii][kk] != NULL$) then
            #pragma omp task input(A[kk][kk])
            bdiv($A[kk][kk]$, $A[ii][kk]$);
        end
    end
    for ($ii = kk + 1; ii < NB; ii++$) do
        for ($jj = kk + 1; jj < NB; jj++$) do
            if ($A[kk][jj] != NULL$) then
                if ($A[ii][jj] == NULL$) then
                    $A[ii][jj] = allocate\_clean\_block()$;
                end
                #pragma omp task input(A[ii][kk], A[kk][jj])
                bmod($A[ii][kk]$, $A[kk][jj]$, $A[ii][jj]$);
            end
        end
    end
end
#pragma omp taskwait
```

and corruption in the program output as follows:

$$RE(c_1, c_2) = \frac{overhead(c_2)}{overhead(c_1)} \times \frac{errors(c_2)}{errors(c_1)} \tag{2}$$

where $overhead(c*)$ is the time overhead of the redundant execution and $errors(c*)$ is the number of corrupted elements in the final output when the component $c*$ is set to be the risk of the task and our heuristic is enabled. If $RE(c_1, c_2) > 1$, then c_1 is more effective than c_2. Larger value implies c_1 has a clearer advantage in terms of providing good tradeoff between performance and coverage. With this, we proceed as follows: We calculate the relative effectiveness for every pair of components for each benchmark. We take the average for each pair across the benchmarks. Using the order of averages, we sort the components in terms of effectiveness. Then, we normalize averages and calculate weights in the risk formula. Our experiments show that components such as the number of inputs and outputs are almost equally effective as the size of inputs and outputs. Hence, we dismiss them in the risk formula above. For our risk definition, the weights are found to be $w_i = 2.03$, $w_o = 2.71$ and $w_s = 1.32$. In this work, *we only use this method to tune the risk definition.* As future work, we consider to look at the possibility of providing a semi-automated feedback mechanism for our heuristic by utilizing this method and the profiling of applications.

Our motivation, in this study, is to leverage only what is available at runtime, and avoid logging/utilizing high-cost meta-data (for instance task execution times, memory access patterns) which can be only obtained by extensive offline profiling. Not only this, but this meta-data then has to be maintained and identified with the corresponding tasks dynamically at runtime which can be clearly prohibitive due to additional performance, memory and power costs.

Lessons learned from Component Analysis:

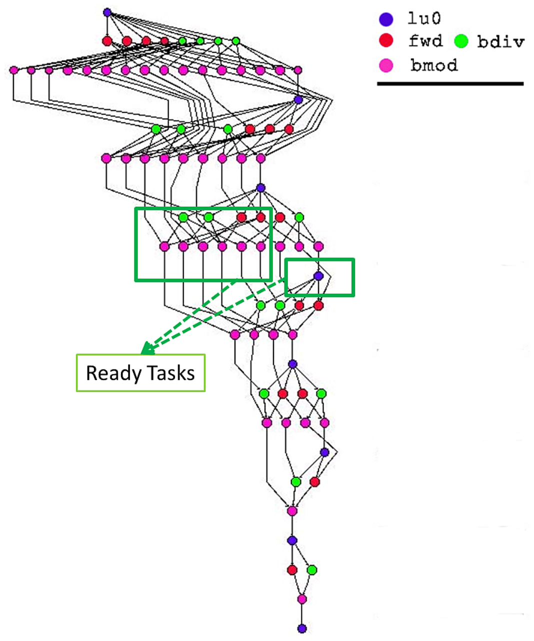

Figure 2: Sparse LU Task Dependency Graph

Benchmark	Most Effective Components or Tasks
Sparse LU	Size or number of arguments
Cholesky	Number of immediate successors
FFT	Early tasks
Perlin Noise	Late tasks
Stream	No outstanding component or task

Table 1: Lessons learned from Component Analysis

We now summarize our findings from the component analysis (see also Table 1):

1.) The size of arguments and the number of arguments are nearly equally effective across all benchmarks. We suspect that this due to the fact that the program input is evenly distributed among the tasks.

2.) In Stream, other components lead to the protection of all tasks since they are the same for all tasks.

3.) In Sparse LU, the size of arguments (or equally number of arguments) are more effective than other components (by about 2×). In addition, the number of immediate successors is more effective than the number of immediate predecessors (by about 3×).

4.) In Cholesky, the number of immediate successors is the most effective component.

5.) For FFT and Perlin, certain tasks (early and late tasks respectively) are more effective in preventing the data corruption without regard to different components. In FFT and Perlin, there is only one unique static task definition. Thus, individual task components, such as the size of inputs or the number of immediate successors, become indistinguishable in terms of being more effective than the other components. Instead, individual task instances that are executed earlier or later during the benchmark execution become more distinctive and effective than the complementary tasks.

Sparse LU	LU decomposition Matrix size 6400x6400, block size 100x100
Cholesky	Cholesky factorization Matrix size 16384x16384 and block size 512x512
FFT	Fast Fourier Transform Array size 16384x16384, block size 16384x128
Perlin Noise	Noise generation to improve realism in motion pictures Array of pixels with size of 65536, block size 2048
Stream	Linear operations among arrays Array size 2048x2048 (doubles), block size 32678

Table 2: Details of shared-memory benchmarks

Nbody	Interaction between N bodies Array size 65536 bodies, block size depends on #nodes
Matmul	Matrix Multiplication using CBLAS Matrix size 9216x9216 doubles and block size 1024x1024
Pingpong	Computation and communication between pairs of processes Array size 65536 doubles, block size 1024
Linpack	HPL Linpack ported to OmpSs Matrix size 131072 doubles, block size 256, 8x8 grid

Table 3: Details of OmpSs+MPI benchmarks

4 EVALUATION

We run our experiments in a production supercomputer. The overall system summary is as follows: The total number of cores is 48,896 and the total number of compute nodes is 3,056. Each node has two Intel SandyBridge-EP E5-2670 processors with 8-cores at 2.6 GHz and 32 GB DDR3 memory (16 cores per node). The supercomputer has 100.8 TB of main memory, 2 PB disk storage, interconnection network Infiniband FDR10 and Linux SuSe OS.

Our task-parallel benchmarks are of two-fold: Pure shared-memory OmpSs (Table 2) and hybrid OmpSs+MPI benchmarks adopting the OmpSs model (Table 3). For the former, we have four HPC benchmarks, - Sparse LU Decomposition, Cholesky Factorization, Fast Fourier Transform, Perlin Noise - and an artificial synthetic Stream benchmark [16]. We conduct experiments to assess the cost of task redundancy in terms of scalability and performance. We obtain all results by running each single case 10 times and take the average of overheads, speedups or execution times. The OmpSs+MPI benchmarks are used to evaluate the scalability and overheads of the task redundancy at large-scale.

We evaluate our selective redundancy mechanism in terms of effectiveness in detecting SDCs. For our mechanism, we take the averages of percentages of detected and corrected errors and of the final output corruptions as the final results. We measure the final output corruption by comparing the corresponding elements in the correct outputs of the fault-free executions to those in the outputs of the fault-injected executions. The injection overheads are included in the reported overheads. We next present the complete and selective task redundancy results in Section 4.1 and 4.2 respectively.

4.1 Complete Task Redundancy Results

In this section, we evaluate the scalability and overheads of complete task redundancy. Our motivation is that if experiments show that complete task redundancy scales and has low overhead, then it trivially follows that partial redundancy also scales and has low overheads. In addition, the scalability experiments will indicate whether task redundancy have any negative effect on the scalability of the applications.

Figure 3 illustrates the strong scalability of the complete task redundancy for shared memory benchmarks. In the figure, the x-axis shows the number of cores and y-axis shows the speedups. They show the baseline/original speedups without any fault injection and any redundancy, speedups without any fault injection but with complete redundancy (0% fault rate), and with 20% and 40% fault rate per task fault injections with complete redundancy. As seen, complete task redundancy scales very well, which means it does not affect scalability negatively. In Stream, we see that it scales well up to 4 cores (the super-linear speedup with 4 cores is due to the fact that task arguments fit in 20 MB caches of each core). It does not scale with 8 and 16 cores even when task redundancy is disabled. Stream mostly consists of memory operations and does not have much parallelism. We purposely chose it to analyze the behavior of an artificial memory-intensive benchmark in terms of performance and scalability since it is a good candidate to stress task redundancy.

We note that the fault rates used for injections are high considering the fault frequencies in HPC systems since the rates that we experiment with are for a *single* task. This in effect simulates higher fault rates for the overall system. Our motivation is that if we achieve good performance for the upper bound case of high error rates, it would confirm our expectation that our implementation is what we call “failure scalable".

Figure 4 shows the weak scalability of complete task redundancy for Sparse LU and Stream. The x-axis shows the number of cores and y-axis shows the execution time. As we can see, for Sparse LU, when the application input size is increased proportional to the number of cores, the execution time stays roughly constant in all fault rates (including some noise) which means it is weakly scaling. However Stream is not weakly scaling due to reasons stated above. Complete task redundancy is also weakly scaling for Cholesky, Perlin Noise and FFT. For brevity, those results are not shown.

Figure 6(a) shows the complete task redundancy performance overheads with respect to the fault-free execution times of the benchmarks. These overheads include task input checkpointing and output comparison overheads when benchmarks are executed with 16 cores. However for other core counts they are similar. As seen, they are very low for Sparse LU, Cholesky, Perlin and modest for FFT and Stream. In FFT with the given benchmark block size, the task granularity is very coarse; there are only 256 application tasks. This prevents effective overlapping of checkpointing with task computations. In Stream, memory system is further stressed with the additional I/O due to checkpointing. Note that duplicate tasks are executed on spare cores.

OmpSs+MPI Applications Results: We include and evaluate these hybrid MPI and task parallel applications in order to experiment and show i) complete task redundancy is well-suited for such hybrid MPI programs and ii) it incurs low overhead and is highly scalable for high core counts at large scale. Figure 5 shows the scalability of the complete task redundancy for OmpSs+MPI benchmarks i.e. speedups over 64 cores. We see that complete task redundancy is also scaling for these benchmarks. Moreover, Figure 6(b) shows the performance overheads with respect to the fault-free execution times when 1024 cores are used. As we can see, the overheads are low.

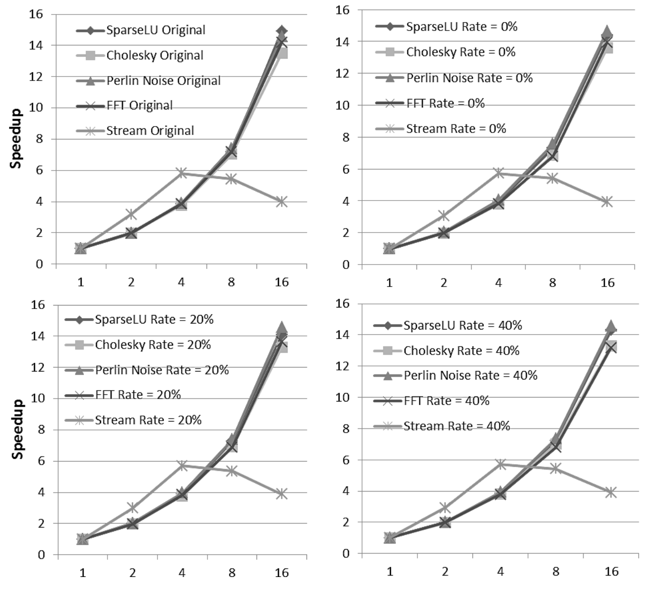

Figure 3: Strong Scalability in Shared memory

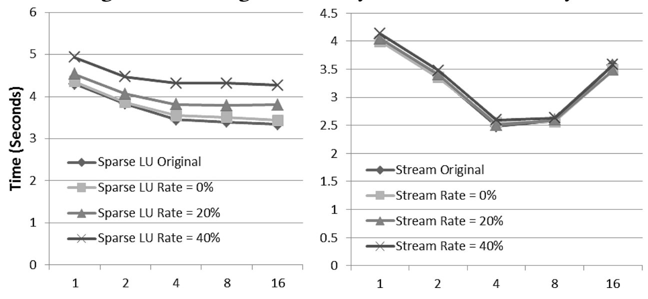

Figure 4: Weak Scalability: Sparse LU and Stream

4.2 Selective Task Redundancy Results

To assess the selective redundancy framework thoroughly, we use two fault injection settings. First with the fixed fault-rate per task injection setting, we aim to analyze the performance of the framework in a task-centric manner. Second with a fault injection setting based on the application execution time and input size, we aim to assess it in an application-centric manner, that is to consider the memory space and the time in the processor pipeline when the data is alive. Including the task execution time in the injection model enables us to mimic the impact of combinational logic errors in the processor pipeline. We have two different scenarios for the second injection setting: Time 1 is the one where we inject 1 fault per 1 second of the fault-free execution time per 32 MBs of the benchmark input. For the case of Perlin Noise, we inject 1 fault per 1 second per 2048 pixels. In Time 2, we inject at a rate of around 1/5 of Time 1. We use these two time-based settings to analyze our framework in different fault occurrence probabilities.

Tables 4 A) and B) show the automated task protection results of our heuristic by showing the percentage of the performance

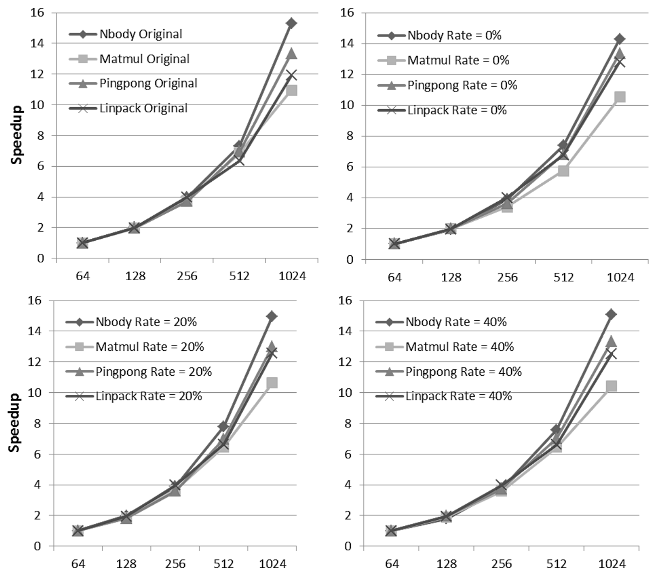

Figure 5: Strong Scalability in OmpSs+MPI

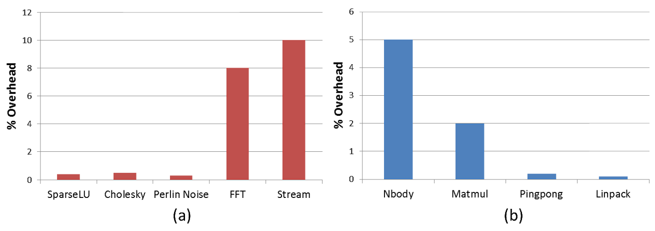

Figure 6: Task redundancy overheads

A) Automated Task Selection: Fixed Rate Injection				
	Fault Rate 3%		Fault Rate 20%	
Benchmarks	**Overhead**	**% DetectCorrect.**	**Overhead**	**% DetectCorrect.**
SparseLU	1.8%	38%	3.5%	18%
Cholesky	0.6%	57%	4.7%	23%
FFT	2.9%	99%	8%	99%
Perlin	1.4%	75%	10%	38%
Stream	3.1%	77%	3.6%	80%
B) Automated Task Selection: Time-based Injection				
	Time 1		Time 2	
Benchmarks	**Overhead**	**% DetectCorrect.**	**Overhead**	**% DetectCorrect.**
SparseLU	1.8%	39%	0.9%	89%
Cholesky	1.7%	49%	1.2%	46%
FFT	27.7%	99%	21.9%	99%
Perlin	0.07%	100%	0.02%	100%
Stream	3.6%	84%	2.4%	99%

Table 4: Selective Task Selection Results

overhead including re-execution of tasks to the fault-free execution time and the percentage of detected and corrected errors. The results for FFT and Stream show that our heuristic is very effective. For Cholesky and Sparse LU, it is moderately effective. We note that in the time-based scenario we see that FFT has relatively more overhead (more than 20%) than other benchmarks. The reason is that around half of the tasks are selected to be protected and this overhead comes from the re-execution overhead of these selected long-running tasks. For Perlin Noise, the results for fixed-rate injections are similar to the results of Cholesky and Sparse LU. In case of time-based injections, since the number of injections is significantly lower than those of fixed-rate injections, we almost always detect and correct errors.

We additionally compare our heuristic to the programmer-directed task selection (user-specified) [18] and to random task selection to discuss their effectiveness. Figure 7 shows how effective our heuristic and the programmer hints are. In all benchmarks, both our heuristic and the programmer hints are more effective than the random selection. However, note that while the programmer knowledge based method requires expert application knowledge, the risk based method does not. We experimentally obtain random task selection curve by running experiments with different random task selection rate and 3% per task fault injection rate. We then construct the linear curve (by extrapolation), which represents the independent tasks, by calculating successive points assuming that the percentage of corruption is linearly correlated with the percentage of task protection. We start from the first point where fault injection is performed with no task protection. This conceptual curve is drawn to see the case as if the tasks were completely independent.

We now analyze the effect of the task dependencies in our applications on the error propagation to the final output. Obviously, in an application when tasks are independent of each other, then the final output corruption is a linear function of the percentage of task being protected (represented by the linear curve). However, because of the dataflow execution model, tasks are dependent on each other. Thus, the final output corruption is a nontrivial function of the percentage of task protection and this function differs for each particular application. For Sparse LU and Cholesky, we see that this function is similar and different than the linear case. For Stream, even though there are task dependencies, the application behavior is almost same as if the tasks were independent. For FFT and Perlin Noise, the application behavior shows that there are some tasks that are clearly crucial than others in terms of final output corruption.

5 RELATED WORK

System-wide fault-tolerance proposals targeting SDC or fail-stop errors such as [7, 10] are orthogonal and seamlessly integrable to our framework for complete error coverage. Fiala et al. [10] provide SDC mitigation for MPI applications in which they provide both consistency and SDC detection and correction protocols for MPI communications. Chung et al.[7] introduce the Containment Domains (CDs) which are programming constructs for the programmer to express resilience concerns and to specialize fault detection and recovery according to target application. Compared to CDs, we provide a completely transparent and fully automatic runtime heuristic that selectively protects the most reliability-critical sections of applications.

Selective redundancy was previously proposed to improve the responsiveness of a distributed system by using the spare resources for replication [12] and to only protect the most critical or vulnerable parts of the code by executing them on reliability enhanced hardware components or replicating them on available computational resources [5].

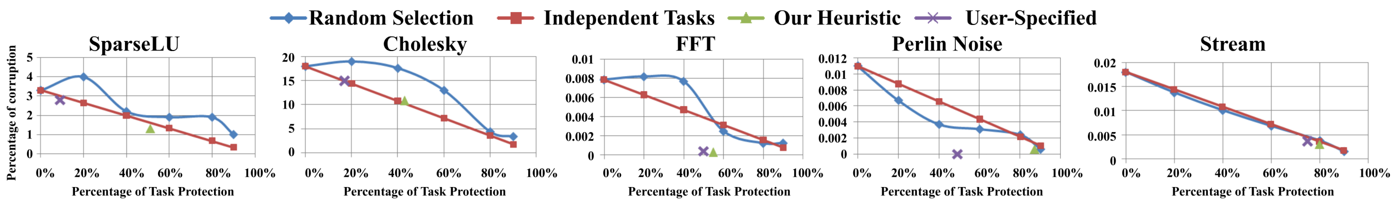

Figure 7: Comparison to Random Task Selection

Subasi et al. [19, 20] propose elaborate and heavyweight partial redundancy schemes for decreasing the performance, power and energy costs. However, the techniques provided are still expensive where the performance overheads of the proposed heuristics, for instance, can be higher than 10% whereas our heuristic has negligible performance overheads. Moreover, these techniques differ in terms of how they achieve partial redundancy. Both techniques use quotas - target replication percentage or target failure rates - to realize partial replication. In particular, [19] is based on a Markov model where the redundancy for a single task is varying over the lifetime, which is different from our technique.

6 CONCLUSIONS

In this work, we present our partial task redundancy framework implemented in a publicly available programming model and its dataflow runtime to mitigate silent and fail-stop errors for task-parallel HPC applications. We show why task-based dataflow programming is the right environment to provide resiliency solutions. We show that our design is highly efficient and scalable. We provide an automatic risk-based heuristic which is shown to be effective by our results for the cases where partial fault-tolerance is considered adequate.

In this work, our research findings are the following:

- Our risk-based heuristic is effective and at the same time highly lightweight.
- Our technique is as good as the programmer-directed method [18] and performs better than random task selection. We note that our risk-based method does not require expert application knowledge whereas the programmer-directed method does.
- Our heuristic mitigates the silent and fail-stop errors simultaneously and incurs negligible performance overhead.
- Component analysis shows that different applications have different reliability-important components or tasks.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number 66905, program manager Lucy Nowell. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830 and the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357.

REFERENCES

[1] Intel TBB 4.3 Update 2. http://www.threadingbuildingblocks.org/documentation.
[2] Abdelhalim Amer, Naoya Maruyama, Miquel Pericàs, Kenjiro Taura, Rio Yokota, and Satoshi Matsuoka. Fork-Join and Data-Driven Execution Models on Multi-core Architectures: Case Study of the FMM. In *ISC13*.
[3] M. Andersch, C. Chi, and B. Juurlink. Using OpenMP superscalar for parallelization of embedded and consumer applications. In *SAMOS12*.
[4] M. Andersch, B. Juurlink, and C. Chi. A Benchmark Suite for Evaluating Parallel Programming Models. In *PARS11*.
[5] Patrick G. Bridges, Kurt B. Ferreira, Michael A. Heroux, and Mark Hoemmen. Fault-tolerant linear solvers via selective reliability. In *CoRR12*.
[6] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward Exascale Resilience. In *IJPCA09*.
[7] Jinsuk Chung, Ikhwan Lee, Michael Sullivan, Jee Ho Ryoo, Dong Wan Kim, Doe Hyun Yoon, Larry Kaplan, and Mattan Erez. Containment domains: a scalable, efficient, and flexible resilience scheme for exascale systems. In *SC12*.
[8] Jack Dongarra, Pete Beckman, and Terry Moore et al. The International Exascale Software Project Roadmap. In *IJPCA11*.
[9] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters* (2011).
[10] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *SC12*.
[11] Priyanka Ghosh, Yonghong Yan, Deepak Eachempati, and Barbara Chapman. A Prototype Implementation of OpenMP Task Dependency Support. In *IWOMP13*.
[12] Franz Josef Grunberger, Thomas Heinze, and Pascal Felber. Adaptive Selective Replication for Complex Event Processing Systems. In *WBDDD13*.
[13] Yixin Luo, Sriram Govindan, Bikash Sharma, Mark Santaniello, Justin Meza, Aman Kansal, Jie Liu, Badriddine Khessib, Kushagra Vaid, and Onur Mutlu. Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory. In *DSN14*.
[14] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Overlapping Communication and Computation by Using a Hybrid MPI/SMPSs Approach. In *ICS10*.
[15] Bernd Panzer-Steindel. April 2007. Data integrity. In *CERN/IT Draft 1.3 8*.
[16] BSC Application Repository. https://pm.bsc.es/projects/bar/wiki/Applications.
[17] M. Snir, R.W. Wisniewski, and J. A. Abraham et al. Addressing Failures in Exascale Computing. In *IJHPC 2013*.
[18] Omer Subasi, Javier Arias Moreno, Osman S. Unsal, Jesús Labarta, and Adrián Cristal. 2015. Programmer-directed partial redundancy for resilient HPC. In *Proceedings of the 12th ACM International Conference on Computing Frontiers, CF'15, Ischia, Italy, May 18-21, 2015*. 47:1–47:2.
[19] Omer Subasi, Gulay Yalcin, Ferad Zyulkyarov, Osman S. Unsal, and Jesús Labarta. 2016. A Runtime Heuristic to Selectively Replicate Tasks for Application-Specific Reliability Targets. In *2016 IEEE International Conference on Cluster Computing, CLUSTER 2016, Taipei, Taiwan, September 12-16, 2016*. 498–505.
[20] Omer Subasi, Gulay Yalcin, Ferad Zyulkyarov, Osman S. Unsal, and Jesús Labarta. 2017. Designing and Modelling Selective Replication for Fault-tolerant HPC Applications. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain, May 14-17, 2017*. 452–457.
[21] Xavier Teruel, Xavier Martorell, Alejandro Duran, Roger Ferrer, and Eduard Ayguadé. Support for OpenMP tasks in Nanos v4. In *CASCON07*.