

Lecture 21: Resilience in the Exascale Era

Vivek Kumar

Computer Science and Engineering

IIT Delhi

vivekk@iiitd.ac.in

Last Lecture (Recap)

- False sharing
- Runtime solutions for detecting/repairing false sharing

Today's Class

- ➔ ● Exascale computing
- Approaches for resilience
- Runtime solutions for resilient task-parallel programs

What is Exascale Computing Project?

- When we say the Exascale Computing Project – what comes to mind?
 - Hardware / systems / platforms?
 - Software / software stack?
 - Applications?
- If you were thinking “all the above” – you were right

Achieving *capable* exascale computing

- Support applications solving science problems 50× faster or more complex than today's 20 PF systems
- Operate in a power envelope of 20–30 MW
- Be sufficiently resilient (average fault rate no worse than weekly)
- At least two diverse system architectures
- Possess a software stack that meets the needs of a broad spectrum of applications
- A holistic project approach is needed that uses co-design to develop new platform, software, and computational science capabilities at heretofore unseen scale
 - Essential for tackling much deeper challenges than those that can be solved by hardware scale alone

Key Challenges for Exascale

- Parallelism
 - Main highlight of this course
- Memory and Storage
 - We covered NUMA and locality in context of the memory, but we are not covering storage in this course
- Reliability
- Energy Consumption
 - Covered in lectures 17 and 18

Today's Class

- Exascale computing
- ➔ ● Approaches for resilience
- Runtime solutions for resilient task-parallel programs

Resilience

- It is the techniques for keeping applications running to a correct solution in a timely and efficient manner despite underlying system faults
- System faults lead to failures, where the system provides an incorrect service (e.g., crashes or provides wrong answers)
- Resilience is a major roadblock for HPC executions on future exascale systems
 - These systems will typically gather millions of CPU cores running up to a billion threads.

The Exascale Resilience Problem

- Hardware failures

- Expected to be more frequent
 - A system 1,000 times more powerful will have at least 1,000 times more components and will fail 1,000 times more frequently


- Software failures

- As hardware becomes more complex (heterogeneous cores, deep memory hierarchies, complex topologies, etc.), system software will become more complex and hence more error-prone
- Increase use of open source layers means less coordinated design in software, which will increase the potential for software errors
- As per some research, large parallel jobs may fail as frequently as once every 30 minutes on exascale platforms

Approaches for Resilience (1/5)

- Checkpointing

- Rollback recovery approach using checkpoint/restart
- The programmer adds some specific functions in the application to save essential state and restore from this state in case of failure
 - Drawbacks
 - Non-optimal placement of checkpointing code
- I/O could become a bottleneck while writing large amount of intermediate program state in the disk
- Checkpointing steps taken to avoid IO latency
 1. The checkpoint is first stored in local SSD (faster than disk). It supports process failure but not node failure
 2. Store the checkpointing in the remote SSD to support single node failure
 3. Support multinode failure by breaking down the checkpointing into blocks and distributing it to multiple nodes
 4. Store the checkpointing in parallel file system to support catastrophic failures such as full system outage
- Frequency (granularity) of checkpointing could control the latency



European MPI Users' Group Meeting
↳ EuroMPI 2010: **Recent Advances in the Message Passing Interface** pp 219–228 | [Cite as](#)

Checkpoint/Restart-Enabled Parallel Debugging

[Joshua Hursey](#), [Chris January](#), [Mark O'Connor](#), [Paul H. Hargrove](#), [David Lecomber](#), [Jeffrey M. Squyres](#) & [Andrew Lumsdaine](#)

Conference paper
938 Accesses | **4** Citations

Part of the [Lecture Notes in Computer Science](#) book series (LNCS, volume 6305)

Abstract

Debugging is often the most time consuming part of software development. HPC applications prolong the debugging process by adding more processes interacting in dynamic ways for longer periods of time. Checkpoint/restart-enabled parallel debugging returns the developer to an intermediate state closer to the bug. This focuses the debugging process, saving developers considerable amounts of time, but requires parallel debuggers cooperating with MPI implementations and checkpointers. This paper presents a design specification for such a cooperative relationship. Additionally, this paper discusses the application of this design to the GDB and DDT debuggers, Open MPI, and BLCR projects.

Download book PDF

Sections | **References**

- [Abstract](#)
- [References](#)
- [Author information](#)
- [Editor information](#)
- [Rights and permissions](#)
- [Copyright information](#)
- [About this paper](#)

- Checkpoint restart also used for debugging parallel programs, but why?

Approaches for Resilience (2/5)

- Forward recovery

- In some cases, the application can handle the error and execute some actions to terminate cleanly or follow some specific recovery procedure without relying on classic rollback recovery
 - Any example?
 - You can correlate it with handling some checked-type exceptions in Java (e.g., `IOException`, etc.), where the checkpointing happens inside the exception handler code (*finally* block) so that the application can be restarted at the same stage by fixing the `IOException`
 - If there is no support for checkpoint/restart, the long running application must be restarted from the beginning!
- A prerequisite for rollforward recovery is that some application processes and the runtime environment stay alive
 - In the above example, the JVM could stay alive to complete the checkpointing upon failure

Approaches for Resilience (3/5)

● Replication

- Each process is replicated such that the probability that all replicas would fail is acceptably small
- Replicas of a process are assigned to different computers
- They proceed asynchronously with the same code and data such that they can be viewed as an integrated logical entity by others
- A distributed computation should proceed correctly as long as there exists one living replica for each process.
- Drawback
 - Amount of computational resources is a major challenge
 - Usually double the number of resources actually required by the program

Approaches for Resilience (4/5)

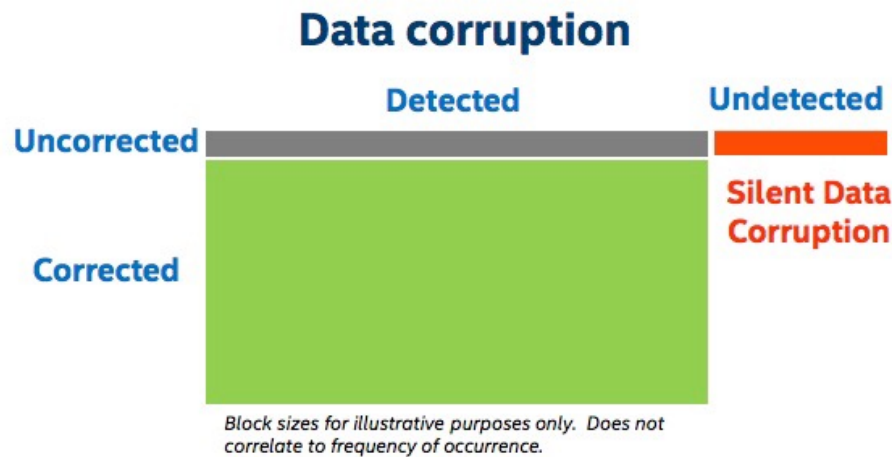
- Failure prediction

- Draw conclusions about upcoming failures from the occurrence of previous failures
 - Measure the system behaviour using hardware metrics, and compare it to the expected normal behaviour using some machine learning algorithms
 - Some errors can be predicted by their side-effects on the system such as exceptional memory usage, CPU load, disk I/O, or unusual function calls in the system
 - Periodically measure such system in order to identify an imminent failure

Silent Data Corruption

Likely to be the most important type of fault in the exascale systems

“When data corruption goes undetected, it becomes silent and is a high risk for applications” EnterpriseStorageForum.com



SDC events may occur more frequently than perceived

 **11** drives per 1000 can experience SDC in a year¹

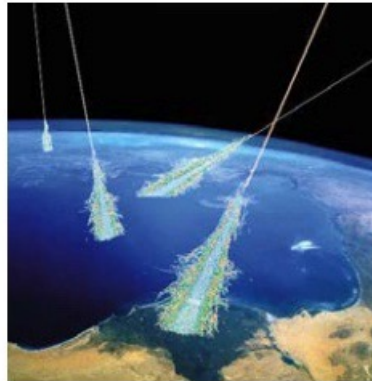
up to 10% catastrophic storage system failures have been linked to SDC¹

See appendix for footnote 1

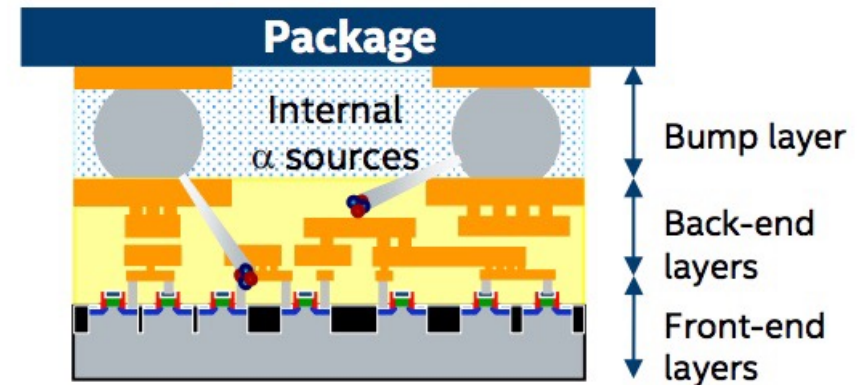
Silent Data Corruption has two main external causes

Cosmic Rays

1. Protons and heavy ions originate from the sun and stars
2. They interact with atmosphere creating neutrons
3. Neutrons multiply quickly in a cascading reaction
4. At earth's surface, ~10 neutrons/sec. pass through a person



Alpha particles



1. Trace radioactive elements exist in some materials
2. Even pure leads can generate 1 particle/cm²/khr.

Approaches for Resilience (5/5)

- Mitigating Silent Data Corruption (SDC) or Soft Errors
 - Industry-wide hardware issue impacting computer CPUs
 - An SDC occurs when an impacted CPU inadvertently causes errors in the data it processes
 - For example, an impacted CPU might miscalculate data (i.e., $1+1=3$) due to manufacturing defects
 - The transistors are so tiny that small electrical fluctuations can cause errors
 - Online solutions to test a new processor
 - <https://github.com/google/cpu-check>

Today's Class

- Exascale computing
- Approaches for resilience
- ➔ ● Runtime solutions for resilient task-parallel programs

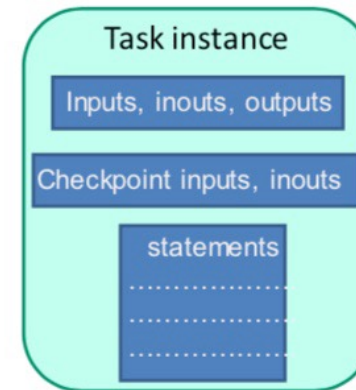
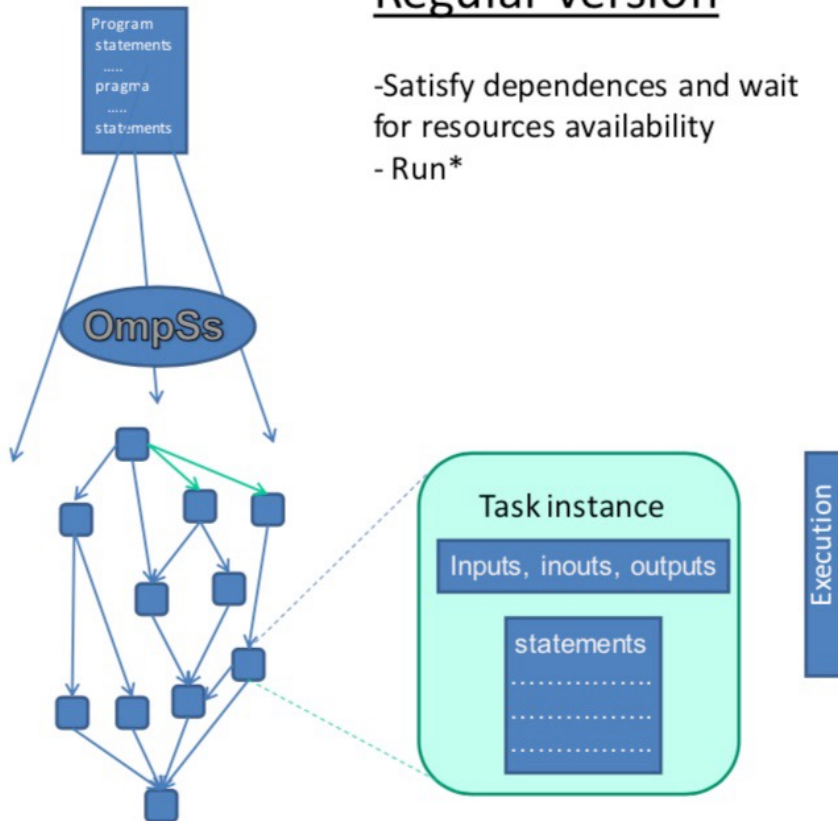
Checkpoint Restart in OmpSs

Regular version

- Satisfy dependences and wait for resources availability
- Run*

Checkpoint Restart version

- Satisfy dependences and wait for resources availability
- Checkpoint inputs, inouts to checkpoint structure
- do {
 - if(fail) Restore checkpoint structure
 - Run*;
 } while (Non-deterministic fail)



**OmpSs
is a task
based
programming
model based
on OpenMP**

Resilience using HClib

- Support resilience using promises and futures
 - Replication
 - Checkpoint and restart
 - Forward recovery (algorithm based fault tolerance)

Resilience using HClib

```

1  auto val1_dep = new promise();
2  auto val2_dep = new promise();
3  auto res_dep = new promise();
4
5  void read_first_val() {
6      async([=] {
7          val1 = new value(get_val_from_src());
8          val1_dep->put(val1);
9      }); // async
10
11 void read_second_val() {
12     async([=] {
13         val2 = new value(get_val_from_src());
14         val2_dep->put(val2);
15     }); // async

```

```

16
17 void operation_val() {
18     async_await ([=] {
19         val1 = get_value(val1_dep);
20         val2 = get_value(val2_dep);
21         res = new value(op(val1, val2));
22         res_dep->put(res);
23     }, val1_dep->get_future()
24        , val2_dep->get_future() ); // async
25 }
26
27 void print_result() {
28     async_await ([=] {
29         res = get_value(res_dep);
30         print(res);
31     }, res_dep->get_future() ); // async
32 }

```

Task Creation
Satisfying a promise
Waiting on a promise

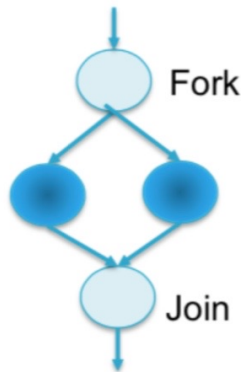
- A baseline non-resilient task parallel program to perform an operation on two asynchronously generated values
 - How to implement the above programming model in the runtime?

Resilience using HCLib

```

1  auto err_dep = new promise();
2
3  void operation_val() {
4      replication::async_await_check ([=] {
5          val1 = get_value(val1_dep);
6          val2 = get_value(val2_dep);
7          res = new value(op(val1, val2));
8          res_dep->put(res)
9      }, err_dep ,
10     val1_dep->get_future(),
11     val2_dep->get_future()); //async
12 }
13
14 void print_result() {
15     async_await([=] {
16         recoverable = get_value( err_dep
17         if (recoverable == 0) exit(1);
18         res = get_value(res_dep);
19         print(res);
20     }, res_dep->get_future(),
21     err_dep->get_future()); // async
22 }

```



The task replication construct
A promise with a failure status

- Resilient task-parallel program based on replication to perform an operation on two asynchronously generated values
- Replicate the task and check for equality of put operations at the end of the task
 - User required to provide equality checking operator for each datatype T used in the promise<T>
 - err_dep promise tells whether a majority of replicas produced the same output
- If error checking succeeds, actual puts are performed
- If error checking fails, puts are ignored and the error is reported using an output promise

Resilience using HClib

```

1  bool  err_chk_func  (void *data) {
2      if (data is good) return true;
3      else return false;
4  }
5
6  auto err_dep = new promise();
7  void *chk_data = nullptr;
8
9  void operation_val() {
10     replay::async_await_check} ([=]{
11         val1 = get_value(val1_dep);
12         val2 = get_value(val2_dep);
13         res = new value(op(val1, val2));
14         res_dep->put(res);
15         chk_data = res;
16     }, err_dep, err_chk_func , chk_data ,
17     val1_dep->get_future(),
18     val2_dep->get_future()); // async
19 }

```

The task replay construct
 User-defined error checking function
 Arguments to the error checking

- Resilient task-parallel program based on replay to perform an operation on two asynchronously generated values
- Instead of applying a rollback of the entire program, as few as one tasks are replayed when an error is detected
- The task is replayed (up to N times) on the original input if its execution resulted in some errors
- Programmer needs to provide an error checking function so that the runtime can use it to check for errors
- The programmer needs to fill the data (chk_data) that needs to be checked for errors using the error checking function (err_chk_func)

Have we ever discussed any FT technique?

- Recall trace/replay?
 - Could be used for achieving resilience
 - How?
 - Is it coarse granular or fine granular approach?

Reference Materials

- Towards exascale resilience
 - <http://snir.cs.illinois.edu/listed/J53.pdf>
- Enabling resilience in asynchronous many-task programming models
 - <https://www.osti.gov/servlets/purl/1641008>
- Exascale vision of India
 - <https://amritmahotsav.negd.in/presentation/day5/Exascale%20Vision%20of%20India.pdf>
- Silent data corruptions at scale
 - <https://arxiv.org/pdf/2102.11245.pdf>

Next Lecture

- Data race detection
- Quiz-4
 - Syllabus: entire course