Leveraging a Task-based Asynchronous Dataflow Substrate for Efficient and Scalable Resiliency

Omer Subasi, Javier Arias, Jesus Labarta, Osman Unsal and Adrian Cristal

Barcelona Supercomputing Center
• Background
  • OmpSs and Nanos
  • Target fault models
• Advantage of our substrate for resilience
• Our proposed solutions
  • Checkpoint restart (detected uncorrected errors (DUE))
  • Task redundancy (silent data corruption (SDC + DUE))
  • Partial redundancy (SDC + DUE)
OmpSs and Nanos

- **OmpSs**—Task based programming model (OpenMP derivative)
  - Task - Once started can execute to completion independent of other tasks
  - Programmer supplies directionality annotations on tasks arguments
- **Nanos**—Runtime supporting OmpSs
  - Dataflow-based - if a task is “ready”, it will be scheduled to a processing element
  - Constructs dataflow graph dynamically from task dependencies

Dependency graph

![Dependency Graph](image)
Our Fault Model From Hardware Failure Taxonomy (by Symptom)

- Undetected
  - Benign (masked faults)
  - Silent Data Corruption
- Hardware Detected
  - Hardware Corrected
  - Hardware Uncorrected
    - Detected Uncorrected Error (DUE)
Outline

• Background
  • OmpSs and Nanos
  • Target fault models
• Advantage of our substrate for resilience
• Our proposed solutions
  • Checkpoint restart (detected uncorrected errors (DUE))
  • Task redundancy (silent data corruption (SDC + DUE))
  • Partial redundancy (SDC + DUE)
Potential and Advantages of OmpSs and Nanos for resilience

- All task inputs and outputs known
  - It is relatively easy and efficient to checkpoint the inouts of tasks
    - facilitates recovery
  - It is relatively easy to replicate tasks and to check the outputs of the replicated tasks
    - facilitates fault detection
- Nanos tasks executed asynchronously and parallel
  - Inherently easy to implement asynchronous and parallel fault tolerance features
- Tasks deferred only because of their dependencies
  - Any redundancy (checkpointing, reissuing) defers only part of execution dependent on it
  - Thus, more efficient than mechanisms subject to fork/join parallelism and than synchronous approaches
Advantages of OmpSs and Nanos for resilience (Cont.)

• Efficient incremental checkpointing schemes easily employed
  • since we only need to checkpoint the inputs of a task

• All dependencies among tasks are known, which
  • facilitates the development of runtime heuristics which can
    determine which tasks are more reliability-critical
  • facilitates partial redundancy
  • Both programmer specified & automated and adaptive

• The only state that propagates out of the task is through the outputs and inouts:
  • straightforward to limit error propagation, and to determine the source of an error
• Background
  • OmpSs and Nanos
  • Target fault models
• Advantage of our substrate for resilience
• Our proposed solutions
  • Checkpoint restart (detected uncorrected errors (DUE))
  • Task redundancy (silent data corruption (SDC + DUE))
  • Partial redundancy (SDC + DUE)
**Checkpoint Restart Implementation**

**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)

*Instance of task is run in parallel within the rest of the task instances
Experimental Setup and Benchmarks

- Experiments run in MareNostrum III (Sandy Bridge)
- Benchmarks
  - Cholesky
    - Matrix size 16384x16384 and block size 512x512
  - Sparse LU
    - Matrix size 6400x6400, block size 100x100
  - Fast Fourier Transform
    - Array size 16384x16384, block size 128
Checkpoint Restart: Cholesky Scalability

Scalability of Checkpoint Restart

Cholesky Benchmark

- Original
- Rate = 0%
- Rate = 5%
- Rate = 10%
- Rate = 20%
- Rate = 40%

Time (seconds)

# Threads

1 2 4 8 16
Checkpoint Restart Implementation: Singleton backups

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

CR version and singleton backups
- Satisfy dependences and wait for resources availability
- Checkpoint inputs, inouts using the concurrent backup handler
- do {
    if(fail) Restore checkpoint using the handler
    Run*
} while (Non-deterministic fail)

*Instance of task is run in parallel within the rest of the task instances
### Singleton Mechanism Results

<table>
<thead>
<tr>
<th></th>
<th>SparseLU</th>
<th>Cholesky</th>
<th>FFT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Checkpoint/Rerestart:</strong> Checkpoint Overhead to Fault-free Exe.Time</td>
<td>2%</td>
<td>6%</td>
<td>9%</td>
</tr>
<tr>
<td><strong>Singleton:</strong> Checkpoint Overhead to Fault-free Exe.Time</td>
<td>0.2%</td>
<td>1%</td>
<td>7%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>SparseLU</th>
<th>Cholesky</th>
<th>FFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gain in X in memory usage</td>
<td>31x</td>
<td>32x</td>
<td>2x</td>
</tr>
</tbody>
</table>
Leveraging OmpSs and Nanos for Fault Tolerance

Feb. 5, 2013

Outline

• Background
  • OmpSs and Nanos
  • Target fault models
• Advantage of our substrate for resilience
• Our proposed solutions
  • Checkpoint restart (detected uncorrected errors (DUE))
  • Task redundancy (silent data corruption (SDC + DUE))
  • Partial redundancy (SDC + DUE)
Leveraging OmpSs and Nanos for Fault Tolerance

Task Redundancy: Current Implementation

Regular version
- Satisfy dependences and wait for resources availability
- Run*

TR version
- Satisfy dependences and wait for resources availability
- Checkpoint inputs, inouts using the concurrent backup handler
- Run* and Run parallel duplicated-run;
- If(different results)
  Restore checkpoint using the handler
  Rerun one instance

*Instance of task is run in parallel within the rest of the task instances
Task Redundancy: SparseLU Scalability

Scalability of Task Replication

SparseLU Benchmark

- Original
- Rate = 5%
- Rate = 10%
- Rate = 20%
- Rate = 40%
Outline

- Background
  - OmpSs and Nanos
  - Target fault models
- Advantage of our substrate for resilience
- Our proposed solutions
  - Checkpoint restart (detected uncorrected errors (DUE))
  - Task redundancy (silent data corruption (SDC + DUE))
  - Partial redundancy (SDC + DUE)
Partial Redundancy

- Partial task replication in Nanos Runtime
- Automated replication
- User-specified replication
- Comparison to random task selection
Automated Partial Redundancy

- Simple Runtime Heuristic:
  - Only replicate reliability-critical tasks
  - Select $t$ for replication if $\text{risk}(t) > \text{global task risk}$
  - Global task risk = $0.7 \times \text{global task risk} + 0.3 \times \text{risk}(t)$
  - $\text{risk}(t) = (i + o)^2 + s$

- **i**: # inputs of the task $t$
- **o**: # outputs of the task $t$
- **s**: # successors of the task $t$
• To capture the memory space used by the tasks as well as dependency among tasks
• Number of inputs/outputs is good hint for memory space usage
• The more a task has successors, the more the severe effect of not protecting the task in terms of error propagation to the successors
User-specified Selective Task Replication

- User specifies which tasks to protect for runtime
- FTT (early tasks)
  - As being a iterative refinement algorithm, early stages likely to be more reliability-critical
- Cholesky (diagonal tasks)
  - As these blocks are utilized during all subsequent phases of the algorithm, directly or indirectly
- SparseLU (no clear distinction between tasks but can protect early tasks processing diagonal elements)
Selective Task Replication Results: FFT

Random Task Selection vs. Our Heuristic vs. User
FFT

Percentage of Incorrect Entries in Output

Percentage of Task Selection

- Random Selection
- User-Specified
- Our Heuristic
Selective Task Replication Results: Cholesky

Random Task Selection vs. Our Heuristic vs. User Cholesky

Percentage of Number of Incorrect Entries in Output

Percentage of Task Selection

- Random Selection
- User-Specified
- Our Heuristic
Conclusions

- OmpSs and Nanos can be leveraged to develop efficient fault-tolerance mechanisms
- Current results seem promising
  - Scalable
  - Low overhead for checkpointing
  - Parallel and asynchronous