Resiliency in massively parallel quantum simulations

R. Solca¹, G. Bosilca², R. Graham³, P. Staar¹, A. Kozhevnikov¹, and Thomas C. Schulthess

(1) ETH Zurich; (2) Univ. of Tennessee; (3) Oak Ridge National Laboratory
Outline

- Introduction
- Monte Carlo integration
- Domain decomposition integration

- first application: DCA++
  - overview
  - results

- second application: DRC
Overview of the problem

- Node failures on massively parallel systems
  - As the number of nodes increases, individual components have to be more reliable (and expensive), or we can try to make the software system, and thus the entire system, resilient to component failure.

- This talk is about creating simulations systems that are resilient to component failure.
What is it that we can expect?

- We have (at least) three possibilities
  - The system catches and corrects errors at a very low level
    - none of the library or application level software has to be aware of resilience
  - Components of the system (e.g. nodes) can fail and the software stack has to catch and deal with the problem at the OS or library level
    - even here there are some fundamental problems – does the application have to know that it is using a fault resilient version of a library?
  - But not all codes use libraries and libraries may not be able to deal with failures, hence application have to deal with problems and recover to be resilient
Two versions of MPI that support fault resilience

- **Fault Tolerant MPI (FT-MPI)**

- **Run-through Stabilization**
Algorithm-based fault tolerance of matrix operations

Outline

- Introduction
- Monte Carlo integration
- Domain decomposition integration
- first application: DCA++
  - overview
  - results
- second application: DRC
Let’s look at a simple problem: Monte Carlo

Algorithm 1 General Monte Carlo algorithm

1: root prepares data
2: Broadcast(data)
3: Compute local results (each process has an independent Markov chain)
4: Allreduce(results)
5: average = results / (processes · measurement per process)
6: output the results

We use collective to find out about the state of the system.
Default broadcast (uses tree algorithm)
Default broadcast – doesn’t work for FT Monte Carlo
Default broadcast – doesn’t work for FT Monte Carlo

0

1

2

3

0

1

2

3

Success

Failure No data

Success
Default allreduce (uses butterfly algorithm)
Default allreduce – doesn’t work either ...

![Diagram showing the default allreduce process with nodes labeled 0, 1, 2, 3, and outcomes Success, Failure, Success, Failure. Red X indicates failure.]
Broadcast that is usable in FT Monte Carlo method

Use functions supplied by FT communication library to check state of system
> MPI_Barrier in FT-MPI
> MPI.Validate_All in run-through stabilization

**Algorithm 2** Fault tolerant broadcast

1: choose a root
2: if this process is the root then
3: prepare data
4: end if
5: repeat
6: Broadcast(data)
7: Check
8: if Failure then
9: Recover
10: if root failed then
11: GoTo 1
12: end if
13: end if
14: until Success

In Algorithm 3 we present the pseudo-code of the modified fault tolerant algorithm that computes the average of the measurements.

**Algorithm 3** Fault tolerant average of results

1: repeat
2: Allreduce(results)
3: Check
4: if Failure then
5: Recover
6: end if
7: until Success
8: average = result / talive processes·measurements per process

Another way to compute an integral in parallel is to decompose the domain between the processes. Each process compute the local part of the integral independently and at the end the local results are summed to get the result.

In Algorithm 4 we show the general domain decomposition algorithm. In contrast with Montecarlo integration some processes failsy the result associated to the domain of the failed processes will not be included in the result. In this case we have to recompute the missing part. Since a code can contain
Allreduce that is usable in FT Monte Carlo method

Algorithm 3 Fault tolerant average of results

1: repeat
2:   Allreduce(results)
3:   Check
4:   if Failure then
5:     Recover
6:   end if
7: until Success
8: average = result / (alive processes \cdot measurements per process)
FT Monte Carlo method

Algorithm 1 General Montecarlo algorithm

1: root prepares data
2: Broadcast(data)
3: Compute local results (each process has an independent Markov chain)
4: Allreduce(results)
5: average = results / (processes · measurement per process)
6: output the results

Cannot use default broadcast and allreduce, use customized versions instead
FT Monte Carlo method: Theoretical analysis

- Montecarlo error with $p$ processes
  \[ \text{err}_p = \frac{\sigma}{\sqrt{n_{tot}}} \propto \frac{1}{\sqrt{n_p p}} \]

- In case of some process failures
  - With FT algorithm the error becomes
  \[ \text{err}_p (1 - f) = \sqrt{\frac{p}{p(1 - f)}} \text{err}_p \approx \left(1 + \frac{f}{2}\right) \text{err}_p \]

- With standard algorithm: NO results
Error behavior in case of failures
Outline

- Introduction
- Monte Carlo integration
- Domain decomposition integration
  - first application: DCA++
    - overview
    - results
  - second application: DRC
Second problem: domain decomposition integration (no data recovery needed)

In the case we do not want to rebuild died processes

Algorithm 1 Domain decomposition

1: root prepares data
2: Broadcast(data)
3: divide the domain among the processes
4: compute the local part of the integral
5: Allreduce(result)
6: output the results

We use collective to find out about the state of the system
FT domain decomposition

First possible way:

**Algorithm 5** Fault tolerant domain decomposition: first way

1: while integration not completed do
2:   divide the remaining interval among the alive processes
3:   compute the local part of the integral
4:   repeat
5:     Allreduce(partial_result)
6:     Check
7:     if Failure then
8:       Recover
9:     end if
10:    until Success
11:  result += partial_result
12: end while

The intervals of the failed processes is divided between the remaining processes

Drawback: need reduction at each cycle
FT domain decomposition

Second possible way:

Algorithm 6 Fault tolerant domain decomposition: second way

1: divide the interval among the alive processes
2: compute the local part of the integral
3: repeat
4: Allreduce(result)
5: Check
6: if Failure then
7: Recover
8: end if
9: until Success
10: for all dead processes do
11: compute the integral over local interval
12: result += local_result
13: end for

The intervals of the failed processes is computed by each process

Drawback: more computation
Outline

- Introduction
- Monte Carlo integration
- Domain decomposition integration

- first application: DCA++
  - overview
  - results

- second application: DRC
Superconductivity: a state of matter with zero electrical resistivity

Heike Kamerlingh Onnes (1853-1926) Discovery 1911

Microscopic Theory 1957

Superconductivity in the cuprates 1986
Quantum cluster theories

Antiferromagnetic correlations / nano-scale gap fluctuations

On-site Coulomb repulsion (~\(\AA\))

Explicitly treat correlations within a localized cluster

Superconductivity (macroscopic)

Treat macroscopic scales within mean-field

Coherently embed cluster into effective medium
Algorithm 1 DCA++

1: root prepares data
2: Broadcast(data) 
3: for \( \text{it} = 1 \rightarrow N_{\text{iterations}} \) do
4: compute some integrals with domain decomposition
5: Monte Carlo integration
6: average results
7: end for
8: output the results

Use previous strategies to include fault tolerance
FT-DCA++ Results

- No performance results:
  - Test on small clusters (20-100 processes per run).

- Self energy after 12 iterations

60 processes

60 processes
30 killed in 30th iteration
Outline

- Introduction
- Monte Carlo integration
- Domain decomposition integration
- first application: DCA++
  - overview
  - results
- second application: DRC
Overview of the application

Bare Coulomb interaction

\[ w(\vec{r}, \vec{r}') = \frac{e^2}{|\vec{r} - \vec{r}'|} \]

Screened Coulomb interaction

\[ W(\vec{r}, \vec{r}', \omega) \]

The challenge: \( W(\vec{r}, \vec{r}', \omega) \) is extremely expensive to compute

Can this be computed at scale and efficiently?
The computationally intensive part – lots of nested loops

\[ \chi_{KK'}(q, \omega) = \frac{1}{N_k \Omega} \sum BZ \sum_k \langle \psi_{j\beta} | e^{i(G+q)r} | \psi_{j'\beta} \rangle \times \]

\[ \frac{f_{j\beta} - f_{j'\beta}}{\epsilon_{j\beta} - \epsilon_{j'\beta} + \omega + i0^+} \langle \psi_{j'\beta} | e^{-i(G'+q)r} | \psi_{j\beta} \rangle \]

\[ \chi_{GG'}(q, \omega) = \frac{1}{N_k \Omega} \sum BZ \sum_k A_{\beta G}^{k,q} B_{\beta G'}^{k,q}(\omega) \]

Reduce to a complex matrix multiply – BLAS3 zgemm
(code rewrite yields order of magnitude improvement in time to solution)
Parallelize with MPI-Grid

\[ \chi_{GG'}^{KS}(q, \omega) = \frac{1}{Nk_\Omega} \sum_{k} \sum_{\beta} A_{\beta G}^{k, q} B_{\beta G'}^{k, q}(\omega) \]

Execution time of DRC code computing W (and U) for La\textsubscript{2}CuO\textsubscript{4}

<table>
<thead>
<tr>
<th>Number of CPU cores</th>
<th>Execution time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>46656</td>
<td>5991</td>
</tr>
<tr>
<td>93312</td>
<td>4993</td>
</tr>
<tr>
<td>186624</td>
<td>1496</td>
</tr>
</tbody>
</table>

- Run time
- Ideal scaling
Parallelize with MPI-Grid

\[ \chi_{G G'}^{KS}(q, \omega) = \frac{1}{N_k \Omega} \sum_{k} \sum_{\beta} A_{\beta G}^k q B_{\beta G'}^k(\omega) \]

Algorithm 1 General workflow

1: build communicators
2: initialize wave-function
3: partition \( q \)-vectors over \( z \)-direction of the MPI grid
4: partition frequency points over \( x \)-direction of the MPI grid
5: for \( q = 1 \rightarrow N_q^{loc} \) do {loop over local fraction of \( q \)-vectors}
6: partition \( k \)-vectors over \( x \)-direction of the MPI grid
7: partition interband transitions over \( y \)-direction of the MPI grid
8: compute local fraction of matrix elements
9: for \( \omega = 1 \rightarrow N_\omega \) do {loop over all frequency points}
10: compute local fraction of \( \chi_{GG'}^{KS}(\omega, q) \)
11: for each \( xy \)-plane reduce the full \( \chi_{GG'}^{KS}(\omega, q) \) on the process who handles frequency \( \omega \)
12: end for
13: for \( \omega = 1 \rightarrow N_\omega^{loc} \) do {loop over local fraction of frequency points}
14: compute local fraction of \( W_{GG'}(\omega, q) \)
15: compute contribution to \( U_{nn'}^T(\omega) \) matrix
16: end for
17: end for
18: reduce full \( U_{nn'}^T(\omega) \) matrix
19: output \( U_{nn'}^T(\omega) \) matrix to files

The reason we choose only to add processes in the \( x \)-direction is purely practical. One can easily notice that each \( xy \)-plane is independent from the other ones. Only at the end the planes synchronize themselves for the reduction of the matrix \( U \). In the case of an MPI implementation that follows the Runthrough Stabilization Proposal [2], unfortunately there is currently no implementation that also supports recovery modes only one plane will be involved in the recovery, while the others can continue the computation. Only before the reduction of the \( U \) matrix a recovery of the communicators in the \( z \)-direction is needed. We do not add processes in the \( y \)-direction because its dimension is usually less than 1; processes. Hence, if we add one more process to this dimension, the number of extra processes is too large. The new grid will then have two grids of communicators in order to be able to do collective communication between \( ftw \) processes and standard processes, i.e., for checksums and recovery operations, and to do collective operations only between standard processes.
Recover data of failed processes

- Need redundant storage

- Strategy:
  - Extra processes that stores negative checksums

- Example: one extra process

  - Checksum: \( x_{p+1} = - \sum_{i=1}^{p} x_i \)
  - Recovery: \( x_f = - \sum_{\substack{i=1 \atop i \neq f}}^{p+1} x_i \)

  - Can recover only one failure
Change in the MPI Grid

- Add a process in the x-dimension

- Two different grids:
  - computational collectives
  - checksum/recovery collectives

- No extra processes in
  - z-dimension: xy-planes almost independent (reduction at the end)
  - y-dimension: has only 1-10 processes

- Drawback: 2 failed processes in the same x-line
Algorithm 2 Fault tolerant workflow
1: build communicators
2: if process is respawned then
3: get application state
4: end if
5: initialize wave-function
6: partition q-vectors over z-direction of the MPI grid
7: partition frequency points over x-direction of the MPI grid
8: if local $U_{nn}^T(\omega)$ have to be recovered then
9: recover local $U_{nn}^T(\omega)$
10: end if
11: for $q = i_1 \rightarrow N_q^\text{loc}$ do \{loop over local fraction of q-vectors\}
12: partition $k$-vectors over x-direction of the MPI grid
13: partition interband transitions over y-direction of the MPI grid
14: compute matrix elements and $\chi_{GG'}^{KS}(\omega, q)$ \{Algorithm 3\}
15: for $\omega = i_3 \rightarrow N_\omega^\text{loc}$ do \{loop over local fraction of frequency points\}
16: compute local fraction of $W_{GG'}(\omega, q)$
17: compute contribution to $U_{nn}^T(\omega)$ matrix
18: repeat
19: checksum contribution $U_{nn}^T(\omega)$
20: if failure then
21: recover application
22: else
23: update local $U_{nn}^T(\omega)$
24: end if
25: until success
26: end for
27: end for
28: repeat
29: reduce full $U_{nn}^T(\omega)$ matrix
30: if failure then
31: recover application
32: end if
33: until success
34: repeat
35: output $U_{nn}^T(\omega)$ matrix to files
36: if failure then
37: recover application
38: end if
39: until success

Algorithm 3 Fault tolerant computation of matrix elements and $\chi_{GG'}^{KS}(\omega, q)$
1: if matrix elements have to be recovered then
2: recover local fraction of matrix elements
3: else
4: repeat
5: Compute local fraction of matrix elements
6: checksum matrix elements
7: if failure then
8: recover application
9: end if
10: until success
11: end if
12: if $\chi_{GG'}^{KS}(\omega, q)$ have to be recovered then
13: recover $\chi_{GG'}^{KS}(\omega, q)$
14: end if
15: for $\omega = i_2 \rightarrow N_\omega$ do \{loop over all frequency points\}
16: compute local fraction of $\chi_{GG'}^{KS}(\omega, q)$
17: repeat
18: for each $xy$-plane reduce the full $\chi_{GG'}^{KS}(\omega, q)$ on the process who handles frequency $\omega$
19: send $\chi_{GG'}^{KS}(\omega, q)$ to the corresponding ft-process
20: if failure then
21: recover application
22: else
23: ft-processes updates the checksum of $\chi_{GG'}^{KS}(\omega, q)$
24: end if
25: until success
26: end for
27: for $q = i_1 \rightarrow N_q^\text{loc}$ do \{loop over local fraction of q-vectors\}
28: partition $q$-vectors over z-direction of the MPI grid
29: initialize wave-function
30: partition frequency points over x-direction of the MPI grid
31: if local $U_{nn}^T(\omega)$ have to be recovered then
32: recover local $U_{nn}^T(\omega)$
33: end if
34: for $q = i_1 \rightarrow N_q^\text{loc}$ do \{loop over local fraction of q-vectors\}
35: compute matrix elements and $\chi_{GG'}^{KS}(\omega, q)$ \{Algorithm 3\}
36: for $\omega = i_3 \rightarrow N_\omega^\text{loc}$ do \{loop over local fraction of frequency points\}
37: compute local fraction of $W_{GG'}(\omega, q)$
38: compute contribution to $U_{nn}^T(\omega)$ matrix
39: repeat
40: checksum contribution $U_{nn}^T(\omega)$
41: if failure then
42: recover application
43: end if
44: until success
45: end for
46: end for
47: repeat
48: reduce full $U_{nn}^T(\omega)$ matrix
49: if failure then
50: recover application
51: end if
52: until success
53: repeat
54: output $U_{nn}^T(\omega)$ matrix to files
55: if failure then
56: recover application
57: end if
58: end if
59: until success
60: end for
61: call MPI recovery function
62: rebuild all communicators
63: recover data
Summary and conclusion

- Fault resilient (production) versions of DCA++ and DRC

- Monte Carlo based codes (e.g. DCA++):
  - FT-MPI works well, but may not be optimal in terms of performance
  - run-through stabilization is more efficient as the communicators are only fixed, and there is no need to rebuild the communicator

- Distributed matrix based codes (e.g. DRC)
  - run-through stabilization currently doesn’t work, as we need to rebuild processes (generally true for domain decomposed problems)
  - FT-MPI works well, but need to rebuild all the communicators

- We do not yet have performance measurements at scale
  - these will be provided soon as we install fault tolerant version of MPI on larger CSCS machines

- We want our application to produce results, not error codes!
  - All codes and algorithms have to be build with FT in mind from the ground up!
  - We need a standard for FT in MPI sooner rather than later
QUESTIONS / COMMENTS?