Exploiting Redundant Computation in Communication-Avoiding Algorithms for Algorithm-Based Fault Tolerance

HPSC 2016

Camille Coti

LIPN, CNRS UMR 7030, SPC, Université Paris 13

April 9th, 2016
Communication-Avoiding Algorithms

Introduced in 2008 by Demmel et al

- Idea: minimize the number of communications
- Additional computations
- Communications are expensive, flops are not

→ Compute more, communicate less

Exist for *los 3 amigos*: LU, QR, Cholesky
Communication-Avoiding QR

Works by \textbf{panels}:

\[ A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = Q_1 \begin{pmatrix} R_{11} & R_{12} \\ 0 & A_{22}^{1} \end{pmatrix} \]

Then, recursively, work on \( A_{22}^{1} \)...
Communication-Avoiding QR

Works by panels:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = Q_1 \begin{pmatrix} R_{11} & R_{12} \\ 0 & A_{22}^1 \end{pmatrix}$$

Then, recursively, work on $A_{22}^1$...

CAQR algorithm

1. Panel factorization:
   $$\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = Q_1 \begin{pmatrix} R_{11} \\ 0 \end{pmatrix}$$

2. Compact representation:
   $$Q_1 = I - Y_1 T_1 Y_1^T$$

3. Update the trailing matrix:
   $$\left( I - Y_1 T_1 Y_1^T \right) \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix} = \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix} - Y_1 (T_1^T Y_1^T \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix}) = \begin{pmatrix} R_{12} \\ A_{22}^1 \end{pmatrix}$$

4. Continue recursively on the trailing matrix $A_{22}^1$
Panel factorization: key piece of the CAQR algorithm

$\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = Q_1 \begin{pmatrix} R_{11} \\ 0 \end{pmatrix}$

The matrix $\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$ is **tall and skinny**: 
- number of lines $\gg$ number of columns

Specific algorithm to compute the QR factorization of a tall and skinny matrix: **TSQR**
TSQR algorithm

Goal: compute the QR factorization of a matrix $A$:
- $A = QR$
- $A$ is tall and skinny

To compute it in parallel on $P$ processes:
- $M =$ number of lines, $N =$ number of columns
- $M \geq NP$
  $\rightarrow$ at least square matrices on each process

\[
\begin{pmatrix}
A_1 \\
A_2 \\
A_3 \\
A_4
\end{pmatrix} = Q_1 \begin{pmatrix}
R_1 \\
0 \\
0 \\
0
\end{pmatrix}
\]
TSQR algorithm

The TSQR algorithm is used to compute the QR decomposition of a matrix. The algorithm is divided into four parts, each conducted by a different process: $P_0$, $P_1$, $P_2$, and $P_3$. The matrices $A_0$, $A_1$, $A_2$, and $A_3$ are computed and sent to the corresponding processes. The resulting matrices $R_0$, $R_1$, $R_2$, and $R_3$ are then used to compute the QR decomposition of the original matrix.

The diagram shows the flow of data and computation between the processes. Each process sends and receives data as indicated by the dashed lines.

The processes perform the following steps:

1. **$P_0$**: Computes $A_0$ and sends it to $P_1$.
2. **$P_1$**: Receives $A_0$ from $P_0$, computes $R_1$, and sends it to $P_2$.
3. **$P_2$**: Receives $A_2$ from $P_3$, receives $R_1$ from $P_1$, and computes $R_2$.
4. **$P_3$**: Receives $A_3$ from $P_3$, computes $R_3$, and sends it to $P_2$.

The resulting matrices $R_0$, $R_1$, $R_2$, and $R_3$ are used to construct the QR decomposition of the original matrix $A$.
TSQR algorithm

\[ P_0 \quad A_0 \quad R_0 \quad V_0 \quad R_0' \quad V_0' \\
\]

\[ P_1 \quad A_1 \quad R_1 \quad V_1 \quad R_1' \quad V_1' \\
\]

\[ P_2 \quad A_2 \quad R_2 \quad V_2 \quad R_2' \quad V_2' \\
\]

\[ P_3 \quad A_3 \quad R_3 \quad V_3 \quad R_3' \quad V_3' \\
\]
### TSQR algorithm

![Diagram of TSQR algorithm](image)
TSQR algorithm

Complexity of the TSQR algorithm:
- Matrix $A$: $M$ lines, $N$ columns; $P$ processes
- $\frac{4}{3} \frac{MN^2}{P} + \frac{3}{4} N^3 \log P$ flops
- $\log P$ communications

Complexity of a traditional QR factorization (ScaLAPACK):
- $\frac{4}{3} \frac{MN^2}{P}$ flops
- $N \log P$ communications

→ Number of communications: save a factor $N$
→ Flops: extra $\frac{3}{4} N^3 \log P$ flops

Compute more, communicate less!
Reliability of a distributed system

Mean Time Between Failures

\[ MTBF_{total} = \left( \sum_{i=0}^{n-1} \frac{1}{MTBF_i} \right)^{-1} \]  

\[ (1) \]

→ The more components a system is made of, the more likely it is to have a failure.
Approaches for fault tolerance: automatic

Automatic fault tolerance:
- Rollback recovery
- Distributed snapshots with coordinated checkpointing (Chandy-Lamport)
- Non-coordinated checkpointing with message-logging
Approaches for fault tolerance: automatic

Automatic fault tolerance:
- Rollback recovery
- Distributed snapshots with coordinated checkpointing (Chandy-Lamport)
- Non-coordinated checkpointing with message-logging

Benefits:
- Completely automatic, transparent
- No modification in the code of the parallel program

Drawbacks:
- Performance overhead: when checkpoints are taken, when messages are logged
- Failure/restart: expensive
  - Coordinated checkpointing: all the processes roll back
  - Non-coordinated checkpointing: only the failed process rolls back, but subsequent synchronizations?
Approaches for fault tolerance: algorithm-based

Behavior upon failures: handled by the application itself

- Failure recovery and sustainability is handled by the parallel program
- Written by the programmer
- Data redundancy, diskless checkpointing
- Iterative checkpointing
- *User-Level Failure Mitigation* (MPI-3 standard)
Approaches for fault tolerance: algorithm-based

Behavior upon failures: handled by the application itself
- Failure recovery and sustainability is handled by the parallel program
- Written by the programmer
- Data redundancy, diskless checkpointing
- Iterative checkpointing
- User-Level Failure Mitigation (MPI-3 standard)

Benefits:
- FT mechanism adapted to the application
- Smaller checkpoints
- Adapted synchronizations

Drawbacks:
- Requires some work from the programmer
- Need for a parallel library and run-time environment that support the ABFT (FT-MPI, MPI-3)
1 Introduction
   • Communication-Avoiding Algorithms
   • Fault tolerance

2 Fault-tolerant TSQR
   • Redundant TSQR
   • Replace TSQR
   • Self-Healing TSQR

3 Performance overhead

4 Conclusion
Let's look at TSQR again

\[ QR \]

\[
P_0 \quad A_0 \quad \rightarrow \quad R_0 \quad V_0
\]

\[
P_1 \quad A_1 \quad \rightarrow \quad R_1 \quad V_1
\]

\[
P_2 \quad A_2 \quad \rightarrow \quad R_2 \quad V_2
\]

\[
P_3 \quad A_3 \quad \rightarrow \quad R_3 \quad V_3
\]
Fault tolerant TSQR

Let’s look at TSQR again

\[ P_0 \rightarrow A_0 \rightarrow R_0 \rightarrow V_0 \rightarrow R_0' \rightarrow P_0' \]

\[ P_1 \rightarrow A_1 \rightarrow R_1 \rightarrow V_1 \rightarrow R_2 \rightarrow V_2 \rightarrow R_2' \rightarrow V_2' \]

\[ P_2 \rightarrow A_2 \rightarrow R_2 \rightarrow V_2 \rightarrow R_3 \rightarrow V_3 \rightarrow R_3' \rightarrow V_3' \]

\[ P_3 \rightarrow A_3 \rightarrow R_3 \rightarrow V_3 \rightarrow R_3' \rightarrow V_3' \]
Fault tolerant TSQR

Let’s look at TSQR again

\[
\begin{align*}
P_0 & \quad A_0 & \quad R_0 & \quad V_0 & \quad R_0' & \quad V_0' & \quad R_0'' & \quad V_0''' & \quad R \\
P_1 & \quad A_1 & \quad R_1 & \quad V_1 & \quad R_1' & \quad V_1' & \quad R_1'' & \quad V_1''' & \quad R' \\
P_2 & \quad A_2 & \quad R_2 & \quad V_2 & \quad R_2' & \quad V_2' & \quad R_2'' & \quad V_2''' & \quad R'' \\
P_3 & \quad A_3 & \quad R_3 & \quad V_3 & \quad R_3' & \quad V_3' & \quad R_3'' & \quad V_3''' & \quad R''' \\
\end{align*}
\]
Let’s look at TSQR again

- $P_0$ works beginning $\rightarrow$ end
- $P_2$ works during the first two steps, then stops
- $P_1$ and $P_3$ work during the first step, then stops

Let’s put these lazy dudes to work!
What do we expect from fault tolerance?

Have **one result** and the end

- No matter how many processes survive, one of them has the final answer
- Here: *Redundant TSQR*
What do we expect from fault tolerance?

Have **one result** and the end
- No matter how many processes survive, one of them has the final answer
- Here: *Redundant TSQR*

Have the result **on a given process** at the end
- No matter how many processes survive, the one we want has the final answer
- Here: *Replace TSQR*
What do we expect from fault tolerance?

Have **one result** and the end
- No matter how many processes survive, one of them has the final answer
- Here: *Redundant TSQR*

Have the result **on a given process** at the end
- No matter how many processes survive, the one we want has the final answer
- Here: *Replace TSQR*

Have the result on the expected process and **all the processes are alive**
- Finish with a system that looks as if nothing bad happened
- Here: *Self-Healing TSQR*
Fault Tolerant TSQR: redundant TSQR

Introduce redundancy between processes: exchange between pairs.

\[ QR \]

\[ \begin{array}{c}
P_0 & A_0 & R_0 \\
A_1 & P_1 & A_2 & R_1 \\
& P_2 & A_3 & R_2 \\
& & R_3 \\
V_0 & V_1 & V_2 & V_3
\end{array} \]
Fault Tolerant TSQR: redundant TSQR

Introduce redundancy between processes: exchange between pairs.

\[ QR \quad Send/Recv \]

\[
\begin{align*}
P_0 & \quad A_0 \rightarrow R_0 \rightarrow R_0 \\
P_1 & \quad A_1 \rightarrow R_1 \rightarrow R_1 \\
P_2 & \quad A_2 \rightarrow R_2 \rightarrow R_2 \\
P_3 & \quad A_3 \rightarrow R_3 \rightarrow R_3
\end{align*}
\]
Fault Tolerant TSQR: redundant TSQR

Introduce redundancy between processes: exchange between pairs.

\[ QR \rightarrow \text{Send/Recv} \rightarrow QR \]

\[
\begin{align*}
P_0 & \quad A_0 & \quad R_0 \rightarrow V_0 \rightarrow R'_0 \rightarrow V'_0 \\
P_1 & \quad A_1 & \quad R_1 \rightarrow V_1 \rightarrow R'_0 \rightarrow V'_0 \\
P_2 & \quad A_2 & \quad R_2 \rightarrow V_2 \rightarrow R'_2 \rightarrow V'_2 \\
P_3 & \quad A_3 & \quad R_3 \rightarrow V_3 \rightarrow R'_2 \rightarrow V'_2
\end{align*}
\]
Fault Tolerant TSQR: redundant TSQR

Introduce redundancy between processes: exchange between pairs.

\[
\begin{align*}
\text{Send/Recv} & : R_0 \rightarrow R_1 & R_1 \rightarrow R_0 \\
\text{Send/Recv} & : R_0' \rightarrow R_2' & R_2' \rightarrow R_0'
\end{align*}
\]
Fault Tolerant TSQR: redundant TSQR

Introduce redundancy between processes: exchange between pairs.
Redundant TSQR: failure

If a process fails: the other ones can continue, except those who need to communicate with the failed process.
Fault Tolerant TSQR: Replace TSQR

When a process fails, another one takes its place: $P_1$ acts as $P_2$. 

\[ QR \quad Send/Recv \quad QR \quad Send/Recv \quad QR \]

\[
\begin{align*}
P_0 & \quad A_0 & \quad R_0 & \quad V_0 & \quad R_0 & \quad V_0' & \quad R_0' & \quad V' \\
P_1 & \quad A_1 & \quad R_1 & \quad V_1 & \quad R_1 & \quad V_1' & \quad R_1' & \quad V_1' \\
P_2 & \quad A_2 & \quad R_2 & \quad V_2 & \quad R_2 & \quad V_2' & \quad R_2' & \quad V_2' \\
P_3 & \quad A_3 & \quad R_3 & \quad V_3 & \quad R_3 & \quad V_3' & \quad R_3' & \quad V_3' \\
\end{align*}
\]
Fault Tolerant TSQR: Self-healing TSQR

Spawn a new process that recovers the data from a twin process

\[
\begin{align*}
P_0 &: A_0 & R_0 & V_0 \\
P_1 &: A_1 & R_1 & V_1 \\
P_2 &: A_2 & R_2 & V_2 \\
P_3 &: A_3 & R_3 & V_3
\end{align*}
\]
1 Introduction
   • Communication-Avoiding Algorithms
   • Fault tolerance

2 Fault-tolerant TSQR
   • Redundant TSQR
   • Replace TSQR
   • Self-Healing TSQR

3 Performance overhead

4 Conclusion
Performance evaluation: what do we measure?

- Overhead during fault-free execution
  - Very important!
  - Cost of the mechanisms put in place to make the FT possible
  - Here: additional communications
  - Same for the three algorithms
Performance evaluation: what do we measure?

- Overhead during fault-free execution
  - Very important!
  - Cost of the mechanisms put in place to make the FT possible
  - Here: additional communications
  - Same for the three algorithms

- Recovery time
  - Depends on a lot of factors!
  - Failure detection (impossible with asynchronous communications)
  - Recovery made by the RTE (spawn and reconnect a new process)
  - Recovery protocol of the algorithm ← only interesting thing here, but hard to measure independently
64 processes, 64 columns ($P = 64$, $N = 64$)
256 processes, 64 columns ($P = 256, N = 64$)
16 processes, 128 columns \((P = 16, N = 128)\)
Introduction

- Communication-Avoiding Algorithms
- Fault tolerance

Fault-tolerant TSQR

- Redundant TSQR
- Replace TSQR
- Self-Healing TSQR

Performance overhead

Conclusion
Three protocols for fault-tolerant QR factorization of tall-and-skinny matrices
- Cornerstone for general QR factorization
- Three recovery algorithms, one for each semantics

**Scalable FT protocol based on scalable algorithms**

Makes use of new features provided by the MPI-3 standard
- FT API now provided by MPI-3
- *User-Level Failure Mitigation*

Next step:
- Apply this to LU, Cholesky (the other *amigos*)
- FT CAQR for general matrices