NLAFET
Parallel Numerical Linear Algebra for Future Extreme Scale Systems

H2020–FETHPC–2014: GA 671633

D6.7

Prototypes for tiled one-sided factorizations with algorithm-based fault tolerance

April 2019

Document information

Scheduled delivery	2019-04-30
Actual delivery	2019-04-29
Version	2.0
Responsible partner	UNIMAN

Dissemination level

PU — Public

Revision history

Date	Editor	Status	Ver.	Changes
date	Name			
2019-04-02	David Stevens	Draft	0.1	Initial version of document produced.
2019-04-16	David Stevens	Draft	1.0	Revised version.
2019-04-17	David Stevens	Draft	1.1	Version to address review comments
2019-04-25	David Stevens	Final	2.0	Final version for submission

Author(s)

Jack Dongarra (UNIMAN)
David Stevens (UNIMAN)
Mawussi Zounon (UNIMAN)

Internal reviewers

Carl Christian Kjelgaard Mikkelsen (UMU)
Jan Papez (INRIA)

Contributors

George Bosilca (ICL)

Copyright

Acknowledgements

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633.

Table of Contents

List of Figures

1 Introduction

The *Description of Action* document states for deliverable D6.7:

> "*D6.7 Prototypes for tiled one-sided factorizations with algorithm-based fault tolerance*
>
> Prototype software for the Cholesky, LU, and QR factorizations with algorithm-based fault tolerance described in D6.6."

This deliverable is in the context of Task 6.3 and demonstrates a prototype for the algorithm-based fault tolerance (ABFT) techniques that have been described in the NLAFET deliverable D6.6 *Algorithm-based fault tolerance techniques*. Fault tolerance methods aim to provide a mechanism for detecting and correcting errors, such as memory bit-flips, data corruption, or intermittant hardware faults that can be expected to occur during large-scale linear algebra computations [4]. The increasing parallelism of modern HPC architectures inevitably leads to an increase in the frequency of such errors [1]. This is illustrated by results in Table 1, where the mean time between failures (MTBF) can be seen to decrease with as the number of computing cores grows. The scale of HPC systems continues to experience rapid growth; for example the Summit system has almost 2.4 million cores, and mean time to failure on systems of this size can be expected to reduce further. Therefore, in order to enable numerical simulation to produce a correct output, efficient algorithms and strategies for the detection and correction of faults are required. ABFT algorithms are a direct response to mitigate the high fault rates anticipated on large-scale HPC systems, as they exploit algorithmic information in order to minimise the computational cost of detecting and correcting such errors within parallel systems.

Table 1: Overall fault rate on different HPC systems from studies by Gupta et al [3].

System	Best rank	Cores	MTBF (hours)
Jaguar XK6	1st (2009 & 2010)	298,592	8.93
Jaguar XT4	6th (2008)	31,328	36.91
Jaguar XT5	2nd (2008 & 2009)	150,152	22.67
EOS	263th (2015)	12,800	189.04
Titan	1st (2012)	560,640	14.51

For a given matrix A, one-sided matrix factorisations such as Cholesky factorisation ($A = LL^T$), QR factorisation ($A = QR$), LU factorisation ($A = LU$) are the innermost kernels for solving systems of linear equations $Ax = b$. These algorithms are time consuming, and therefore can be considered more likely to experience faults. We explain how ABFT strategies can be exploited to design resilient matrix factorisation algorithms. For the sake of generality and simplicity, we will consider $A = ZU$ to represent a one-sided matrix factorisation, where Z is the left matrix and U is an upper triangular matrix. It is also beneficial to consider a one-sided factorisation as recursively applying Z_i to the initial matrix A from the left until $Z_i Z_{i-1} \dots Z_0 A$ becomes upper triangular. In our ABFT implementation we augment the matrix using a two-line checksum in order to identify and correct faults (see Section 2 for more detail). The following theorem then becomes the key ingredient behind the success of ABFT strategies for matrix operations.

Theorem 1.1. *Checksum relationship established before ZU factorisation is maintained during and after factorisation.*

Proof. For a given matrix $A \in \mathbb{R}^{n\times n}$, let $A = ZU$ denote its one-sided factorisation, where $Z \in \mathbb{R}^{n\times n}$, and U is an upper triangular matrix. We denote $A_c = [A, \overline{A}] \in \mathbb{R}^{n\times n+\gamma}$, the original matrix augmented by the row-wise checksum matrix $\overline{A} \in \mathbb{R}^{n\times\gamma}$, where γ is the width of the checksum. In the ABFT algorithm, the factorisation operations are applied to the matrix A_c. If we rewrite U as $Z_n Z_{n-1} \dots Z_0 A = U$, it follows that $\forall i \in [1, n]$

$$\begin{aligned} Z_i Z_{i-1} \dots Z_0 A_c &= Z_i Z_{i-1} \dots Z_0 [A, \overline{A}] && (1)\\ &= [Z_i Z_{i-1} \dots Z_0 A,\ Z_n Z_{n-1} \dots Z_0 \overline{A}] && (2)\\ &= [U^i, \overline{U}^i]. && (3) \end{aligned}$$

Since $Z_i Z_{i-1} \dots Z_0$ is a linear operation, the initial relation between the input matrix A and the corresponding matrix $\overline{A}$ is still preserved in U, and $\overline{U}$ can be considered as the checksum matrix of U. Consequently, at each iteration i of the factorisation, the factor U^i has its corresponding checksum matrix $\overline{U}^i$ that can be used to regenerate any lost portion of U^i. The theorem remains valid in the special case of LU factorisation, so long as each row permutation operation is applied to both the main matrix and the checksum matrix. □

As introduced above, a one-sided matrix factorisation, $A = ZU$, produces two matrices, the left factor Z, and the right factor U. The generic ABFT strategy, described above, protects only U since U results from linear transformation of the initial matrix A, while the left factor Z remains vulnerable to faults. In the special case of Cholesky factorisation ($A = LL^T$), where only one factor (L) is required, ABFT techniques are sufficient to design a fully resilient factorisation algorithm. However, in the case of QR factorisation, Q cannot be protected by ABFT algorithms since the matrix Q is not recursively built by applying linear operator; i.e., Q^{i+1} does not result from a linear transformation of Q^i. The same is true for the left factor L from LU factorisation. Other fault tolerance strategies are required to design a fully resilient QR and LU solvers. Consequently, this deliverable focuses on Cholesky factorisation to demonstrate the potential of ABFT algorithms.

The ABFT prototype software described here uses the PaRSEC framework [2]. The performance of the ABFT approach is demonstrated here using Cholesky factorisation, with a dual-checksum approach, allowing the location of faults to be identified and corrected efficiently.

The structure of this report is as follows: The implementation of the ABFT technique is outlined in Section 2, with instructions for compiling the software described in Section 3. Section 4 provides performance results for the ABFT code in both shared- and distributed-memory environments, followed by concluding remarks in Section 5.

2 Implementation

The ABFT implementation used here operates on a matrix that is partitioned in a tile-based format, and affixes two checksums to each tile (see Figure 1). This tile-based checksum approach allows independent tasks, which operate on individual tiles, to identify and correct faults without the need to access the global matrix. In this way fault correction can be contained within a single task, eliminating the risk of a fault infecting the global solution, and minimising the computational cost of correction.

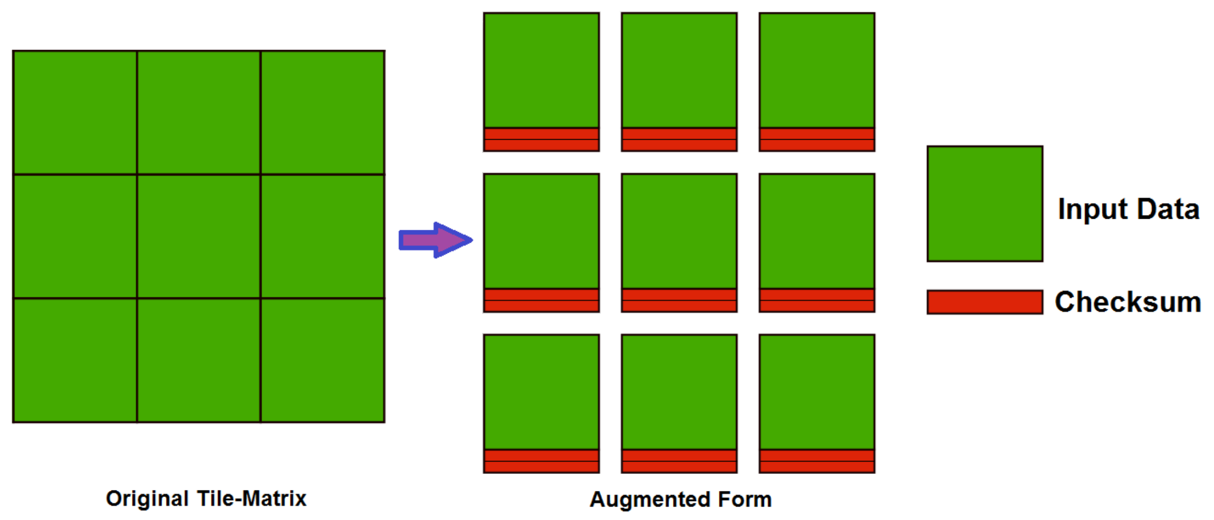

Figure 1: Augmentation of the tile-based matrix with a two-line checksum.

The checksums that augment the matrix are defined as follows:

$$c_1 = e_1\bar{A}, \quad \text{with} \quad e_1 = (1, 1, \dots, 1)^T \tag{4}$$

$$c_2 = e_2\bar{A}, \quad \text{with} \quad e_2 = (1, 2, \dots, nb)^T \tag{5}$$

where nb is the tile size and $\bar{A}$ represents the corresponding matrix tile. To demonstrate the detection and correction procedure we consider an error of magnitude γ at position (i, j) in the matrix tile, i.e.

$$a'_{i,j} = a_{i,j} + \gamma \tag{6}$$

The faulty column may be determined by considering checksum c_1. Column j is faulty if

$$\alpha_1 = \sum_{k=1}^{n} a_{k,j} - (c_1)_j \neq 0 \tag{7}$$

i.e. $\alpha_1 = \gamma$. The faulty row is then identified using checksum c_2, by considering:

$$\alpha_2 = \sum_{k=1}^{n} k\ a_{k,j} - (c_2)j = i\alpha_1 \tag{8}$$

A row i where this does not hold is therefore faulty, and the matrix entry may then be corrected as $a_{i,j} = a'_{i,j} + \alpha_1$.

The addition of the checksum increases the overall matrix size from N to $N(1 + \frac{2}{nb})$. Maintaining the checksum through the Cholesky factorisation procedure therefore leads to a relative computational cost overhead of $(1 + \frac{2}{nb})^3 - 1$.

The detection of faults requires two additional tile-based matrix-vector multiplications for each task performed, at a cost of $2nb^2$ operations. For the $\frac{1}{6}(\frac{n}{nb})^3$ tasks in the tile-based Cholesky routine this translates to $\frac{n^3}{3nb}$ operations in total, representing a relative computational overhead of $\frac{1}{nb}$.

It is important to note that the computational cost and memory overhead of the ABFT routine is dependent only on the tile size chosen. It does not depend on the size of the problem, the number of faults, the type of task that fails or the stage at which the fault occurs.

3 Software compilation

The prototype code is available on the NLAFET GitHub (`https://github.com/NLAFET/ABFT`).

3.1 Requirements

In this section, we provide a brief description of different libraries required to build the ABFT solver.

1. cmake version 2.8.0 or above. cmake can be found in the debian package cmake, or as sources at the cmake download page (http://www.cmake.org/)
2. A BLAS library optimized for your platform: MKL, ACML, Goto, Atlas, VecLib (MAC OS X), or in the worst case the default BLAS (http://www.netlib.org/blas/).
3. hwloc for processor and memory locality features (http://www.open-mpi.org/projects/hwloc/)
4. Any MPI library Open MPI, MPICH2, MVAPICH or any vendor approved implementation.
5. PLASMA version 2.5 up to 2.8 (http://icl.cs.utk.edu/plasma/). Newer versions are not currently supported

3.2 Compilation

The build system is based on CMake. The compilation is quite simple once the dependencies are satisfied. The software can be compiled with the following instructions:

1. Get the source:
   ```
   $ git clone git@github.com:NLAFET/ABFT.git
   ```
2. Navigate to the root directory and create a “build” directory:
   ```
   $ cd ABFT
   $ mkdir build
   $ cd build
   ```
3. Set configuration parameters and execute cmake as in the following bash script:
   ```
   #!/bin/bash
   module load CMake # load cmake
   module load intel/2018.01 # load mkl
   rm -f CMakeCache.txt # remove existing CMakeCache.txt

   # Set compilation parameters and launch cmake
   USER_OPTIONS+=" -DDPLASMA_PRECISIONS=d"
   CC=mpiicc \
   FC=mpiifort \
   CXX=mpiicc \
   cmake ../ \
   -DPLASMA_DIR= [[ PLASMA 2.8 path ]]
   ```

4. Finally, make the application:

```
$ make
```

3.3 Usage

At the end of a successful compilation, different folders will be created within the build directory. From the build directory the testing routines can be found in dplasma/testing:

```
$ cd dplasma/testing
```

From this point the main testing routine is `./testing_dpotrf_abft2`. A help information on the testing usage can be obtained by executing

```
$ ./testing_dpotrf_abft2 --help
```

The fault injection is hard-coded and controlled by the parameter `DO_ERROR` in the file `ABFT/dplasma/lib/zpotrf_L_abft2.jdf` which is a regular PaRSEC JDF file. The verification and the checksum correction are done at the end of each kernel (BODY in the JDF file). The JDF file is easy to modify to control the number of faults to inject, and the kernel in which the fault must be injected. For the sake of comparison the standard task-based Cholesky factorisation without ABFT is also provided with the associated testing routine `./testing_dpotrf_abft` in the testing repository.

4 Results and Testing

The following plots demonstrate the performance of the ABFT algorithm with Cholesky factorisation, using the formulation outlined in Section 2.

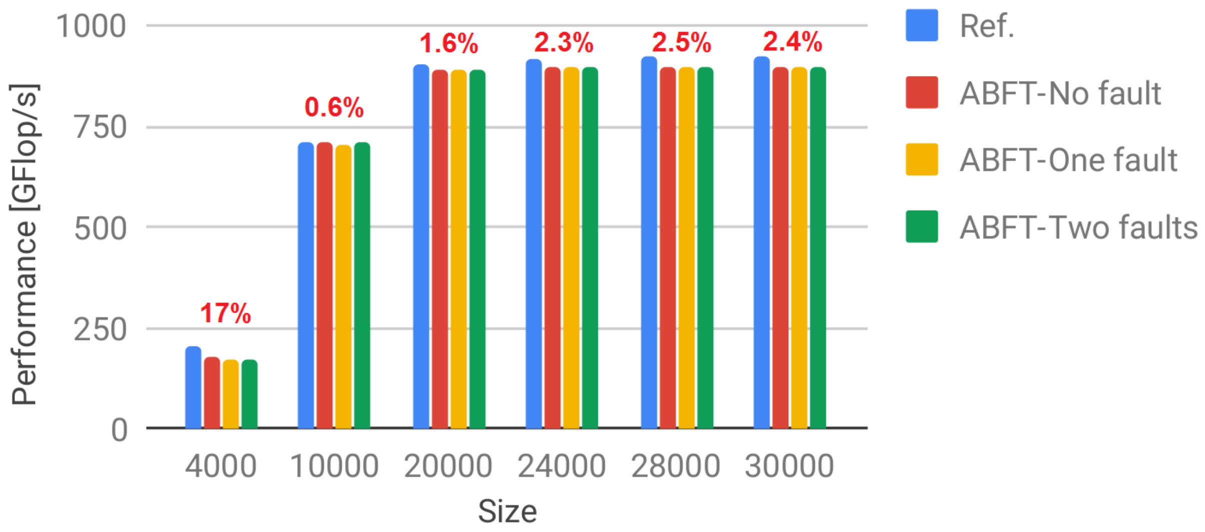

Figure 2: ABFT performance on a single node (Broadwell, 28 cores, 2.6Ghz, MKL 18).

Figure 2 shows the overhead in computational cost of running ABFT on a single node, for varying matrix sizes. The percentages presented for each test show the relative performance loss between the reference solution without ABFT and the slowest of the three ABFT implementations. For all matrices tested the block size is $nb = 256$. The increase in runtime for smaller matrices is moderate, as exhibited by the 17% overhead observed in the $N = 4000$ matrix. However, for larger matrices the performance overhead stabilises to a much smaller value; around 2.5%. The 2.5% overhead observed here aligns well with the theoretical relative increase in FLOP count of $\frac{1}{nb} + (1 + \frac{2}{nb})^3 - 1$ described in Section 2. The results here also demonstrate that the performance overhead is a primarily a result of including the ABFT machinery, rather than from correcting any detected faults; no significant deviation in performance is observed between the cases where no fault, one fault, or two faults occur.

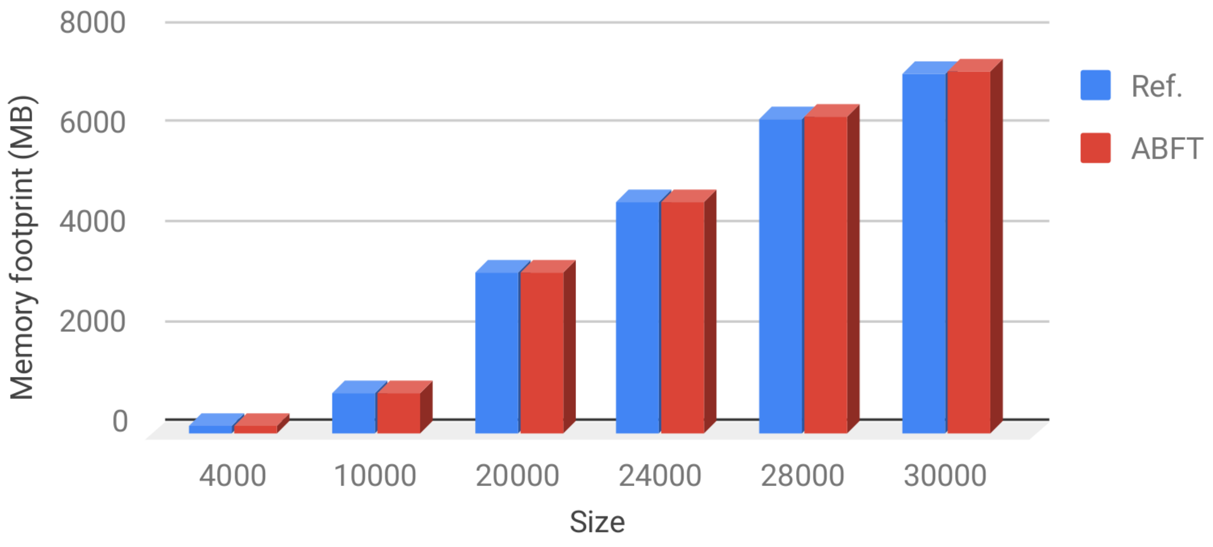

Figure 3: Memory overhead ABFT (Broadwell, 28 cores, 2.6Ghz, MKL 18, single node).

Figure 3 demonstrates the memory overhead observed from the use of ABFT. Here we see that the difference is consistent throughout all matrix sizes at $\frac{2}{nb}$, as expected from the algorithm. For the $nb = 256$ block size used here this translates to a consistent overhead of 0.8%.

Figure 4 shows the performance of ABFT in a distributed memory environment, using nodes of the same type as above, with a matrix fixed at size 80,000. Here we observe a consistent growth of the performance differential as the number of nodes increases, and the corresponding size of the matrix portion placed on each node shrinks. However, the performance overhead remains modest throughout, reaching a maximum of 15% for the 9-node case.

5 Conclusion

This deliverable has provided a prototype implementation of the algorithm-based fault tolerance (ABFT) techniques described in the NLAFET deliverable D6.6. Software for the ABFT-enabled solution of Cholesky factorisation is provided, based on the PaRSEC framework. Performance of the ABFT implementation has been demonstrated within shared-memory and distributed-memory environments. Results show the approach exhibiting a very low memory overhead (<1%), with a small (typically around 2.5%) compu-

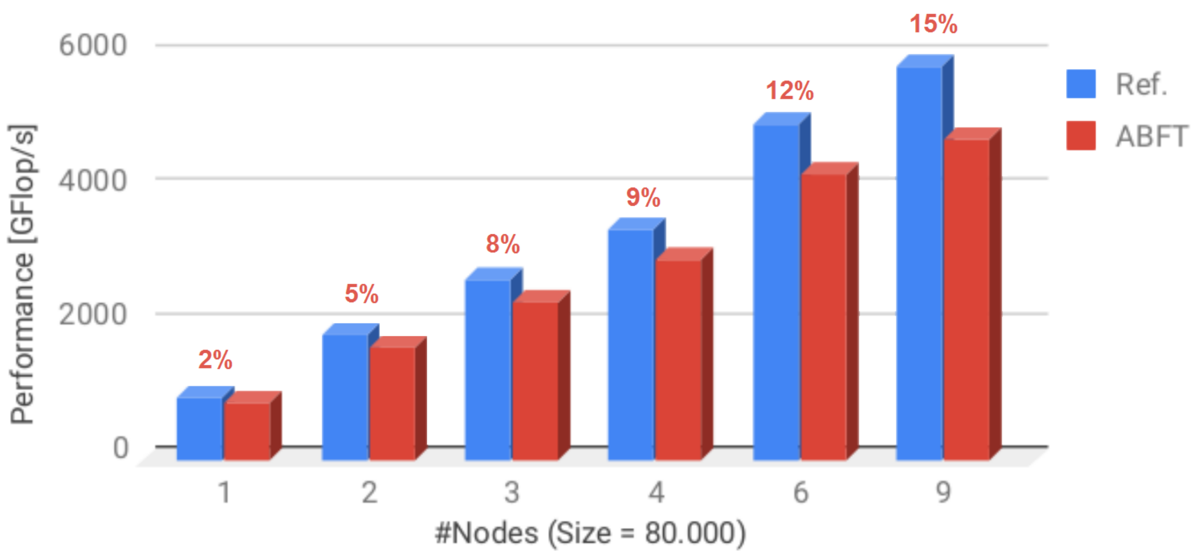

Figure 4: Strong scaling for ABFT in distributed memory environment (each node comprises 28 Broadwell cores at 2.6Ghz).

tational cost overhead for moderate to large sized matrices within shared memory. When using distributed computing the overhead can become somewhat larger, though it remains fairly modest at around 10%. Performance and memory overheads arise from the inclusion of the ABFT framework into the solution, rather than from the correction of any errors discovered, which occurs at negligible computational cost. These findings strengthen the case for use of ABFT as a mechanism for high-performance fault-detection within suitable linear algebra algorithms.

References

[1] George Bosilca, Aurélien Bouteiller, Elisabeth Brunet, Franck Cappello, Jack Dongarra, Amina Guermouche, Thomas Herault, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. Unified model for assessing checkpointing protocols at extreme-scale. *Concurrency and Computation: Practice and Experience*, 26(17):2772–2791, 2014.

[2] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[3] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. Failures in large scale systems: long-term measurement, analysis, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 44. ACM, 2017.

[4] Bianca Schroeder and Garth A. Gibson. A Large-Scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–351, 2010.