Parallel experiments with RARE-BLAS

Chemseddine Chohra, Philippe Langlois and David Parello
Univ. Perpignan Via Domitia,
Digits, Architectures et Logiciels Informatiques,
F-66860, Perpignan.
Univ. Montpellier,
Laboratoire d'Informatique Robotique et de Microélectronique de Montpellier,
UMR 5506, F-34095, Montpellier.
CNRS. France.
mail: firstname.familyname@univ-perp.fr

***Abstract*—Numerical reproducibility failures rise in parallel computation because of the non-associativity of floating-point summation. Optimizations on massively parallel systems dynamically modify the floating-point operation order. Hence, numerical results may change from one run to another. We propose to ensure reproducibility by extending as far as possible the IEEE-754 correct rounding property to larger operation sequences. Our RARE-BLAS (Reproducible, Accurately Rounded and Efficient BLAS) benefits from recent accurate and efficient summation algorithms. Solutions for level 1 (asum, dot and nrm2) and level 2 (gemv) routines are provided. We compare their performance to the Intel MKL library and to other existing reproducible algorithms. For both shared and distributed memory parallel systems, we exhibit an extra-cost of 2× in the worst case scenario, which is satisfying for a wide range of applications. For Intel Xeon Phi accelerator a larger extra-cost (4× to 6×) is observed, which is still helpful at least for debugging and validation.**

I. Introduction and Background

The increasing power of supercomputers leads to a higher amount of floating-point operations to be performed in parallel. The IEEE-754 [1] standard requires the addition operation to be correctly rounded. However because of the errors generated by every addition, the accumulation of more than two floating-point numbers is non-associative. The combination of the non-associativity of floating-point addition and the non-deterministic behavior in parallel programs yields non-reproducible numerical results.

Numerical reproducibility is important for debugging and validating programs. Some solutions are available in parallel programming libraries. Static scheduling and deterministic reduction ensure the numerical reproducibility of the library OpenMP. Nevertheless, the number of threads has to be set for all runs [2]. The CNR feature (Conditional Numerical Reproducibility) has been introduced in Intel MKL library 11.0 release [2]. By limiting the use of instruction set extensions this feature ensures numerical reproducibility between different architectures. Unfortunately, this decreases significantly the performance especially on recent architectures, and requires the number of threads to remain the same from run to run to ensure reproducible results.

First algorithmic solutions are proposed in [3]. Algorithms $ReprodSum$ and $FastReprodSum$ ensure numerical reproducibility independently of the operation order. Therefore, numerical results do not depend on hardware configuration. The performance of these latter is improved with the algorithm $OneReduction$ [4] by relying on indexed floating-point numbers [5] and requiring a single reduction operation to decrease the communication cost. Hence, highly efficient reproducible parallel sum is provided for large distributed memory computing systems. However, those solutions do not improve accuracy. The computed result even being reproducible is still exposed to accuracy problems, especially when an ill-conditioned problem is addressed.

Another way to guarantee reproducibility is to compute accurately rounded results. Recent works [6], [7], [8] show that an accurately rounded floating-point summation can be calculated with very little or even no extra-cost. We have analyzed in [7] different summation algorithms, and identified those suited for an efficient parallel implementation on recent hardware. So parallel algorithms for correctly rounded dot and asum and for a faithfully rounded nrm2 have been designed [7]. This approach has been extended to the matrix-vector multiplication from the level 2 BLAS in [9]. First implementation of level 1 BLAS exhibits interesting performance with 2× extra-cost in the worst case scenario on shared memory parallel systems [7]. Other implementations for distributed memory computing systems and accelerators (Intel Xeon Phi) are introduced in [9].

In this paper, we focus on the experimental part of this work that illustrates the efficiency of our proposed implementation both in terms of run-time and of accuracy. We exhibit their scalability with tests on the Occigen supercomputer. Experiments with the Intel Xeon Phi accelerator illustrate their efficiency and also their portability to many-core systems. Our correctly rounded dot product scales well on distributed memory parallel systems. Compared to optimized but not reproducible implementations, it has no substantial extra-cost up to about 1600 threads (128 sockets, 12 cores). On Intel Xeon Phi accelerator the extra-cost increases up to 6× mainly because our solution benefits less from the high memory bandwidth of this architecture compared to MKLs implementation. Nevertheless, this seems reasonable enough (less than 10×) to be useful for validation, debugging or even for mid-sized applications that could require accurate and reproducible results.

This document is organized as follows. Section II presents our algorithms for reproducible and accurate parallel BLAS. Section III is devoted to implementation and detailed results. Last Section IV presents some conclusions and future work.

II. Parallel RARE BLAS

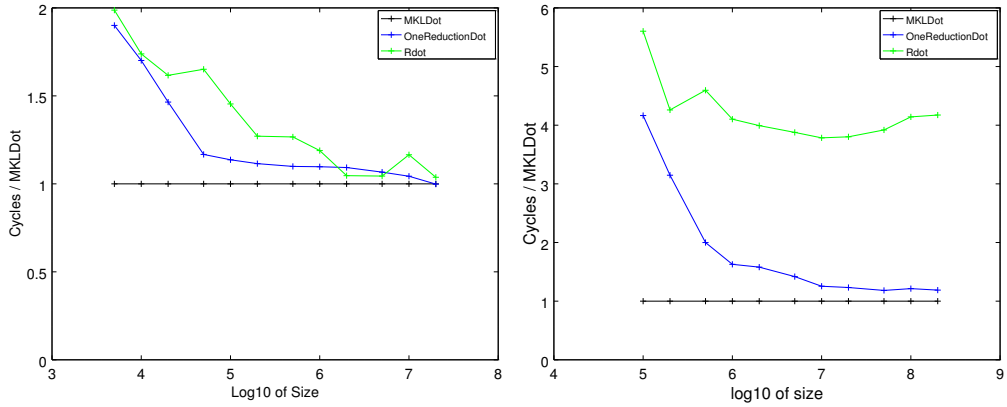
A. Parallel Algorithms for the Level 1 BLAS

1) Sum of Absolute Values: The condition number of $asum$ is known to equal 1. This justifies the use of the algorithm $SumK$ [10]. The parallel version of algorithm $SumK$ [11] is used for parallel $asum$. Two stages are required. (1) The first one consists in applying the sequential $SumK$ on local data without performing the final error compensation. So we end with K floating point numbers per thread. (2) Afterwards the master thread gathers all these numbers in a single vector and applies a sequential $SumK$ on it.

The value of K is picked up such that computing $asum(p)$ as $SumK(|p|)$ is faithfully rounded. Since the condition number equals 1, one appropriate K only depends on the vector size.

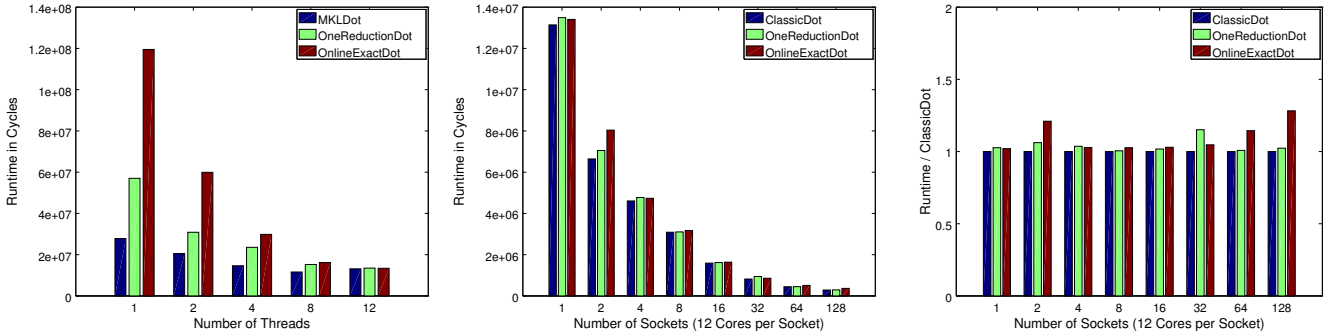
2) Dot Product: Our implementation of parallel reproducible dot and $nrm2$ is presented in [9]. For parallel dot product, we distinguish three steps. (1) In the first step vectors are distributed equally to threads. Each thread make an error-free transformation of its local dot product. Note that the local result is not rounded, the dot product is transformed to a sum of non-overlapping floating-point numbers. The process is done in different ways depending on the vector size (more details in [9]). (2) Afterwards, local results of transformation are gathered by master thread. All previous transformations do not generate any error. (3) Finally the master thread uses $iFastSum$ [12] to calculate a correctly rounded sum of the gathered data (See Figure 1 in [9]).

3) Euclidean Norm: The euclidean norm of a vector p is defined as $(\sum p_i^2)^{1/2}$. The sum $\sum p_i^2$ can be correctly rounded using the previous dot product. Finally, we apply a square root that returns a faithfully rounded euclidean norm [13]. This does not compute a correctly rounded norm-2 but this faithful rounding is reproducible because it depends on a reproducible dot.



(a) Parallel platform A (b) Parallel platform B

Fig. 1: Extra-cost of correctly rounded dot product (cond=10^8)



(a) Single socket (b) Multi socket (c) Multi socket normalized by ClassicDot

Fig. 2: Performance of the distributed memory parallel implementations (platform **C**).

B. Parallel Algorithms for the Level 2 BLAS

Matrix-vector multiplication is defined in the BLAS as $y = \alpha A \cdot x + \beta y$. In the following, we denote $y_i = \alpha a^{(i)} \cdot x + \beta y_i$, where $a^{(i)}$ is the i^{th} row of matrix A.

To compute a correctly rounded value of y_i we pass through the following stages: (1) First we transform the dot product $a^{(i)} \cdot x$ into a sum of non-overlapping floating-point numbers. We use the same algorithm as for parallel dot product in section II-A2. (2) The second step evaluates multiplications by the scalars α and β using $TwoProd$ [14]. Again data are transformed with no error. (3) Finally we distillate results of the previous steps to get a correctly rounded result of $y_i = \alpha a^{(i)} \cdot x + \beta y_i$. The same process should be repeated for each row of the matrix A.

To perform a parallel matrix-vector multiplication let A be a $n-$rows matrix, y be a $n-$elements vector, and let us be given p available threads. The matrix A and the vector y are equally split to threads, and the vector x is shared by all threads. Note that A is split by rows (Figure 2 in [9]). Therefore, each thread will use the previous algorithm to compute the correctly rounded value of n/p elements of y.

This algorithm scales almost perfectly since it does not require any reduction operation.

III. Test and Results

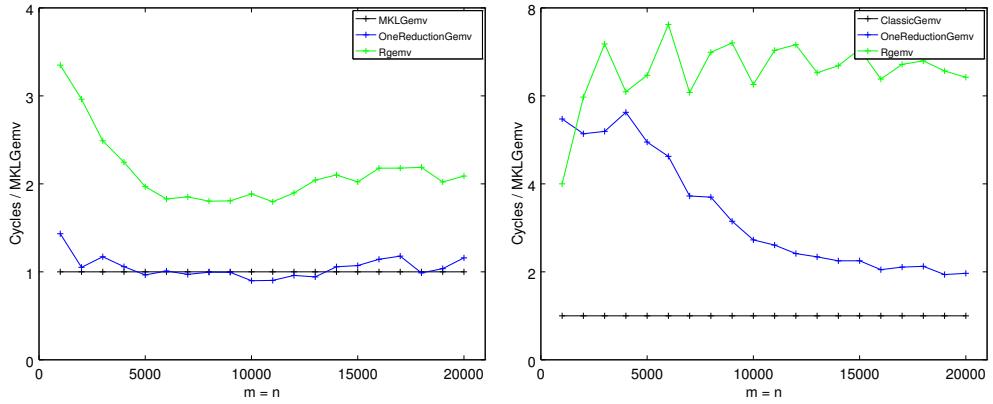
A. Experimental framework

We consider the three frameworks described in Table I. They are significant of today's practice of floating-point computing.

A	Processor	Dual Xeon E5-2650 v2 16 cores (8 per socket), No hyper-threading. L1/L2 = 32/256 KB per core. L3 = shared 20 MB per socket.
	Bandwidth	59,7 GB/s.
	Compiler	Intel ICC 16.0.0.
	Options	-O3 -xHost -fp-model double -fp-model strict -funroll-all-loops.
	Libraries	Intel OpenMP 5. Intel MKL 11.3.
B	Processor	Intel Xeon Phi 7120 accelerator, 60 cores, 4 threads per core. L1/L2 = 32/512 KB per core.
	Bandwidth	352 GB/s.
	Compiler	Intel ICC 16.0.0.
	Options	-O3 -mmic -fp-model double -fp-model strict -funroll-all-loops.
	Libraries	Intel OpenMP 5. Intel MKL 11.3.
C	Processor	Xeon E5-2690 v3 (12 cores per socket), No hyper-threading. L1/L2 = 32/256 KB per core. L3 = shared 30 MB per socket (4212 sockets).
	Bandwidth	68 GB/s.
	Compiler	Intel ICC 15.0.0.
	Options	-O3 -xHost -fp-model double -fp-model strict -funroll-all-loops.
	Libraries	Intel OpenMP 5. Intel MKL 11.2. OpenMPI 1.8.

TABLE I: Experimental frameworks

We test the efficiency of the shared memory parallel implementation on platform **A**. Platform **B** illustrates the many core accelerator use. The scalability of our approach on large supercomputers is exhibited on platform **C** (Occigen supercomputer).



(a) Parallel platform A

(b) Parallel platform B

Fig. 3: Extra-cost of correctly rounded matrix-vector multiplication (cond=10^8)

Data for dot product are generated as in [10]. Data parametrized by the condition number for the matrix-vector multiplication is generated relying on the same algorithm for dot product: matrix A and vector x are such that all dot products of lines $a^{(i)}$ per x verify a condition number given in $[10^5, 10^{16}]$.

B. Implementation and Performance Results

We compare the performance results of our implementation to the highly optimized Intel MKL library, and to implementations based on algorithm $OneReduction$ used by the library ReproBLAS [15]. We have implemented an OpenMP parallel version of this algorithm since ReproBLAS only offers one MPI parallel version. We derive reproducible version of *dot* and *gemv* replacing all non-associative accumulation by the algorithm OneReduction [4]. These versions are for instance denoted $OneReductionDot$ and $OneReductionGemv$. The same implementations is provided for other level 1 BLAS routines.

We do not compare to the CNR feature because it does not guarantee reproducibility between sequential and parallel runs. According to [2] and our tests, the number of threads must remain the same from run to run for the results to be reproducible.

Running time is measured in cycles using the RDTSC instruction. In the parallel case, RDTSC calls have been made out of parallel region before and after function calls.

Parallel implementations for platform **A** are based on OpenMP library. All available 16 cores are used with no hyper-threading. Implementations for the Intel Xeon Phi accelerator are also based on OpenMP. Intrinsic functions are used to benefit from the available instruction set extension AVX-512. A FMA (Fused Multiply and Add) is also available. Therefore $TwoProd$ is replaced by $2MultFMA$ [16] which only requires two FMAs to compute the product and its error, and so improves performance.

In this paper we only focus on $gemv$ and dot. Results for $asum$ and $nrm2$ are presented in [7], [9].

1) Dot Product: Results for dot product are shown in Figures 1 and 2. The condition number of all dot products tested on platforms **A** and **B** is 10^8. In Figure 1a all 3 algorithms $OneReductionDot$, $Rdot$ and $MKLDot$ hit the limit imposed by the memory bandwidth. Therefore, the correctly rounded $Rdot$ do not exhibit any extra-cost compared to $MKLDot$.

Figure 1b shows the performance results on the Intel Xeon Phi accelerator. Since it has much more memory bandwidth than the platform **A**, the $MKLDot$ takes this advantage to improve its scalability. On the other side, our $Rdot$ is not so efficient here because it has many operations that can not be vectorized (but still occupy the 512 bit units to perform scalar operations), which decreases significantly the performance on an accelerator. We measure an extra-cost of $4\times$ in this case.

Finally for dot product we show the performance on a distributed memory parallel system (platform **C**). In this case we have two levels of parallelism: OpenMP is used for thread level parallelism on a single socket and OpenMPI library is used for socket communication. The algorithm scalability is tested on a single data set with input vectors of length 10^7 and a condition number of 10^{32}.

Fig. 2a shows the scalability for a single socket configuration. It is not a surprise that $MKLDot$ does not scale so far since it is quickly limited by the memory bandwidth. $OneReductionDot$ and $OnlineExactDot$ scale well up to exhibit no extra-cost compared to optimized $MKLDot$. Again such scaling occurs until being limited by the memory bandwidth.

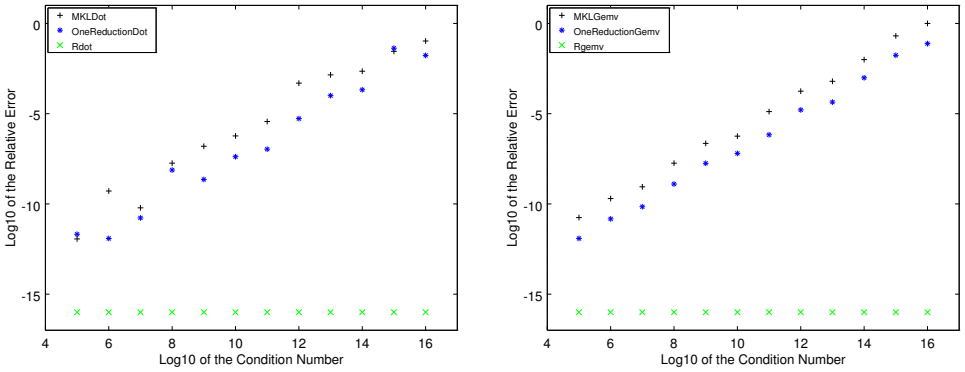
Performance for the multi socket configuration is presented in Figure 2b. $ClassicDot$ denotes a socket local $MKLDot$s followed by a MPI sum reduction. X-axis shows the number of sockets where all the 12 available cores are used. Y-axis represents the execution time and this measure is normalized by $ClassicDot$ on Figure 2c.

Algorithms $OnlineExactDot$ and $OneReductionDot$ are almost as efficient as $ClassicDot$. Since all 12 cores are used per socket all algorithms hit the limit imposed by the memory bandwidth. For inter socket communication all 3 algorithms rely on a single communication. Therefore, they exhibit the same performance on platform **C**.

2) Matrix-Vector Multiplication: Our $Rgemv$ matrix-vector multiplication computes a correctly rounded result using $iFastSum$ [12] for small matrices and $HybridSum$ for large ones, this latter being slightly more efficient than $OnlineExact$ [17] on both platforms **A** and **B**.

In the shared memory parallel case, our correctly rounded $gemv$ costs about twice compared to $MKLGemv$ as shown in figure 3a. As for $MKLDot$, also $MKLGemv$ hits the limit that is imposed by memory bandwidth. Therefore, the scalability of $MKLGemv$ is limited. $Rgemv$ benefits from a better scalability which gives it the advantage of reducing the extra-cost to $2\times$, even if it performs much more floating-point operations.

Intel Xeon Phi results for $gemv$ are presented in Figure 3b. Since the building blocks for both algorithms $MKLGemv$ and $Rgemv$ are



(a) Accuracy results for *dot*

(b) Accuracy results for $gemv$

Fig. 4: Accuracy results for *dot* and $gemv$ (IEEE-754 binary64 with round-to-the-nearest).

respectively based on $MKLDot$ and $Rdot$, we observe the same behavior already noted for the dot product. $MKLGemv$ benefits from higher bandwidth and AVX-512 vector capabilities to run even faster than $Rgemv$. An extra-cost of $6\times$ is observed here.

C. Accuracy Results

Now we present accuracy results for *dot* and $gemv$ variants for the IEEE-754 binary64 arithmetic rounded to the nearest. In both figures 4a and 4b, we measure the relative error with respect to the condition number of the problem. The latter ranges from 10^5 to 10^{16}. Relative errors are calculated comparing with results from MPFR library [18]. The two subroutines $nrm2$ and $asum$ are excluded from this test because condition number is fixed for both of them. In almost all cases, solutions based on algorithm $OneReduction$ besides being reproducible are more accurate than MKL. However, for ill-conditioned problems both MKL and $OneReduction$ derived implementation gives worthless results. On the other side the RARE-BLAS subroutines ensure that results are always correctly rounded independently from the condition number (one horizontal line of plots at the computing precision level).

IV. Conclusion and Future Work

We have introduced algorithms that compute reproducible and accurately rounded results for BLAS. Level 1 and 2 subroutines have been addressed. Implementations of these algorithms have been tested on three platforms significant of the floating-point computing practice. While existing solutions tackle only the reproducibility problem, our proposed solution aims at ensuring both reproducibility and accuracy. We measure interesting performance on CPU based parallel environments. Extra-cost on CPU when all available cores are used is at worst twice compared to optimized but non-reproducible libraries. However performance on Xeon Phi accelerator is lagging behind: extra-cost is between 4 and 6 times more. Nevertheless, our algorithms remain efficient enough to be used for validation or debugging programs, and also for parallel applications that can sacrifice performance to increase the accuracy and the reproducibility of their results.

Our plan for future development includes achieving reproducibility and precision for other BLAS subroutines. We are currently designing an accurate and reproducible version of triangular solver. Other Level 3 BLAS routines will be addressed even if the performance gap with optimized libraries will enforce the previously identified restriction of the application scope.

References

[1] IEEE Task P754, *IEEE 754-2008, Standard for Floating-Point Arithmetic*. New York: Institute of Electrical and Electronics Engineers, Aug. 2008.

[2] R. Todd, "Run-to-Run Numerical Reproducibility with the Intel Math Kernel Library and Intel Composer XE 2013," Intel Corporation, Tech. Rep., 2013.

[3] J. W. Demmel and H. D. Nguyen, "Fast reproducible floating-point summation," in *Proc. 21th IEEE Symposium on Computer Arithmetic*. Austin, Texas, USA, 2013.

[4] ——, "Parallel Reproducible Summation," vol. 64, no. 7, pp. 2060–2070, July 2015.

[5] ——, "Toward hardware support for Reproducible Floating-Point Computation," in *SCAN'2014*, Würzburg, Germany, Sep. 2014.

[6] S. Collange, D. Defour, S. Graillat, and R. Iakimchuk, "Reproducible and Accurate Matrix Multiplication in ExBLAS for High-Performance Computing," in *SCAN'2014*, Würzburg, Germany, 2014.

[7] C. Chohra, P. Langlois, and D. Parello, "Efficiency of Reproducible Level 1 BLAS," Jan. 2015. [Online]. Available: http://hal-lirmm.ccsd.cnrs.fr/lirmm-01101723

[8] R. M. Neal, "Fast exact summation using small and large superaccumulators," *CoRR*, vol. abs/1505.05571, 2015. [Online]. Available: http://arxiv.org/abs/1505.05571

[9] C. Chohra, P. Langlois, and D. Parello, "Reproducible, Accurately Rounded and Efficient BLAS," Feb. 2016, working paper or preprint. [Online]. Available: http://hal-lirmm.ccsd.cnrs.fr/lirmm-01280324

[10] T. Ogita, S. M. Rump, and S. Oishi, "Accurate sum and dot product," *SIAM J. Sci. Comput.*, vol. 26, no. 6, pp. 1955–1988, 2005.

[11] N. Yamanaka, T. Ogita, S. Rump, and S. Oishi, "A parallel algorithm for accurate dot product," *Parallel Comput.*, vol. 34, no. 68, pp. 392 – 410, 2008.

[12] Y.-K. Zhu and W. B. Hayes, "Correct rounding and hybrid approach to exact floating-point summation," *SIAM J. Sci. Comput.*, vol. 31, no. 4, pp. 2981–3001, 2009.

[13] S. Graillat, C. Lauter, P. T. P. Tang, N. Yamanaka, and S. Oishi, "Efficient calculations of faithfully rounded l2-norms of n-vectors," *ACM Trans. Math. Softw.*, vol. 41, no. 4, pp. 24:1–24:20, Oct. 2015. [Online]. Available: http://doi.acm.org/10.1145/2699469

[14] T. J. Dekker, "A floating-point technique for extending the available precision," *Numer. Math.*, vol. 18, pp. 224–242, 1971.

[15] H. D. Nguyen, J. Demmel, and P. Ahrens, "ReproBLAS: Reproducible BLAS," http://bebop.cs.berkeley.edu/reproblas/.

[16] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[17] Y.-K. Zhu and W. B. Hayes, "Algorithm 908: Online exact summation of floating-point streams," *ACM Trans. Math. Software*, vol. 37, no. 3, pp. 37:1–37:13, 2010.

[18] "The MPFR library," URL = `http://www.mpfr.org/`, .