Parallel Implementation of Interval Matrix Multiplication

Nathalie Revol, Philippe Théveny

Parallel Implementation of Interval Matrix Multiplication

Nathalie Revol
INRIA Grenoble - Rhône-Alpes – Université de Lyon
LIP (UMR 5668 CNRS - ENS Lyon - UCB Lyon 1 - INRIA), ENS de Lyon, France
nathalie.revol@ens-lyon.fr

Philippe Théveny
École Normale Supérieure de Lyon – Université de Lyon
LIP (UMR 5668 CNRS - ENS Lyon - UCB Lyon 1 - INRIA), ENS de Lyon, France
philippe.theveny@ens-lyon.fr

Abstract

Two main and not necessarily compatible objectives when implementing the product of two dense matrices with interval coefficients are accuracy and efficiency. In this work, we focus on an implementation on multicore architectures. One direction successfully explored to gain performance in execution time is the representation of intervals by their midpoints and radii rather than the classical representation by endpoints. Computing with the midpoint-radius representation enables the use of optimized floating-point BLAS and consequently the performances benefit from the performances of the BLAS routines. Several variants of interval matrix multiplication have been proposed, that correspond to various trade-offs between accuracy and efficiency, including some efficient ones proposed by Rump in 2012. However, in order to guarantee that the computed result encloses the exact one, these efficient algorithms rely on an assumption on the order of execution of floating-point operations which is not verified by most implementations of BLAS. In this paper, an algorithm for interval matrix product is proposed that verifies this assumption. Furthermore, several optimizations are proposed and the implementation on a multicore architecture compares reasonably well with a non-guaranteed implementation based on MKL, the optimized BLAS of Intel: the overhead is most of the time less than 2 and never exceeds 3. This implementation also exhibits a good scalability.

Keywords: interval arithmetic, interval matrix multiplication, BLAS, gemm, parallel computer, multicore
AMS subject classifications: 65Y05, 65G40, 65F30, 65-04

1 Introduction

When implementing the multiplication of two dense matrices with interval coefficients, two conflicting objectives must be met. A first objective is to get accurate results, i.e. results that do not overestimate too much the exact results, and preferably with known and bounded overestimation ratios. A second objective is to get results quickly, in particular in this work we want to exploit the capabilities of multicore architectures and of the corresponding programming paradigm, namely multithreading. Several algorithms, that offer different trade-offs between these two objectives, are established by Nguyen in [12].

The difficulties one has to face when implementing an interval matrix multiplication are manifold. Implementing interval arithmetic through floating-point arithmetic relies on changes of the rounding modes, either rounding downwards and upwards with the representation by endpoints, or rounding to nearest and upwards with the so-called midpoint-radius representation, using the midpoints and radii. Whether the rounding mode is kept unchanged or modified by BLAS routines is undocumented. Furthermore, whether the rounding mode is properly saved and restored at context switches, as in a multithreaded execution, is not documented either, as pointed out in [8]. Another issue in numerical, floating-point computing, which also impacts the behavior of routines in interval arithmetic, is the non-reproducibility of computations, especially on HPC systems, as noted in [4]. This phenomenon originates in particular from the fact that the floating-point addition is not associative and thus that the order of the additions performed in, for instance, the sum of many numbers depends on the number of threads, the scheduler, the storage in memory of these numbers etc. Thus it is not realistic to rely on any assumption on the order of floating-point computations, which is a problem for the applicability of Theorem 2.1 in Section 2. A last kind of difficulties is related to the programming of multicore architectures. As optimized libraries, such as MKL [6], optimized by Intel for its processors, exhibit good performances, a first idea is to try to use these libraries to benefit from their performances. However, as these libraries do not fulfill any assumption on the execution order of their operations, as mentioned above, they cannot be used and one turns to coding and optimizing the code by hand, using for instance OpenMP and "helping" the compiler to vectorize the code. Getting good performances in the development of such codes is difficult.

In this paper, we focus on algorithms for interval matrix multiplication. Various formulas are based on the midpoint-radius representation, ranging from exact ones as in [11, pp. 22-23] to faster ones in [15] that resort to 4 calls to BLAS floating-point matrix multiplication, including intermediate ones in terms of accuracy and number of calls to BLAS routines given in [12, Chapter 2]. Algorithms that use interval matrix multiplication to verify a floating-point matrix product are given in [14], algorithms that save some floating-point matrix products by bounding the roundoff errors are given in [16]. We will focus on the latter ones. BLAS routines for the product of dense floating-point matrices are detailed in [2]. Optimized versions such as Goto's BLAS [3] or Intel's MKL yield very good performances, autotuned versions such as Atlas [19] yield good performances on various hardware support by tuning automatically cache usage via block size etc.: they are good candidates for interval matrix multiplication algorithms such as those mentioned above, that resort to calls to BLAS floating-point matrix multiplication.

We also focus on architectures that become more and more frequent nowadays and very probably even more in the near future, namely multicore architectures. The machine we use in our experiments is a multiprocessor where each processor is composed of several cores, for even higher performances. This architecture still relies on a shared memory: going to distributed architecture is a major step we did not take, as issues become fundamentally different from shared-memory or sequential ones.

We recall in Section 2 the algorithms from [16] and the assumption on which they rely. As this assumption is no more verified by the implementation of BLAS on the architecture we consider, as explained in Section 3, we give up the use of BLAS. The main result of this paper is **the implementation of dense interval matrix multiplication which is fully guaranteed** as the requirement is fulfilled, **and which gives good performances** as hand-made optimizations are done where needed. It is detailed in Section 4. In particular, exploiting the cache usage through a version by blocks yields performances which are close to the performance of a (non-guaranteed) implementation based on Intel's MKL, in terms of execution time and scalability, as shown in Section 5.

2 Algorithms for Interval Matrix Multiplication

In this paper we use the notation for real interval analysis defined in [7]. A point matrix has its coefficients which are punctual: real or floating-point numbers, as opposed to an interval matrix whose coefficients are intervals. An interval matrix $\boldsymbol{A} \in \mathbb{IR}^{n\times m}$ can be written as a pair of point matrices $[\underline{A}, \overline{A}]$ and represents the set of all matrices $M \in \mathbb{R}^{m\times n}$ such that $\underline{A} \leq M \leq \overline{A}$ where inequalities hold componentwise. Equivalently, the matrix $\boldsymbol{A}$ can be represented by the pair of point matrices $\langle \operatorname{mid} \boldsymbol{A}, \operatorname{rad} \boldsymbol{A} \rangle$ where the (i, j) component of the midpoint matrix $\operatorname{mid} \boldsymbol{A}$ (respectively, of the radius matrix $\operatorname{rad} \boldsymbol{A}$) is the midpoint (respectively, the radius) of $[\underline{A}_{ij}, \overline{A}_{ij}]$:

$$\operatorname{mid} \boldsymbol{A}_{ij} = \frac{\underline{A}_{ij} + \overline{A}_{ij}}{2} \quad \text{and} \quad \operatorname{rad} \boldsymbol{A}_{ij} = \frac{\overline{A}_{ij} - \underline{A}_{ij}}{2}.$$

In [15], Rump showed that the midpoint-radius representation of matrices allows faster matrix operations on computer than their infimum-supremum counterparts. In particular, he described an interval matrix product algorithm that relies on point matrix multiplications for its most expensive computational part. The gain in execution time with this approach comes from the fact that possibly costly rounding mode changes are limited and that numerous implementations of BLAS provide efficient point matrix products. This midpoint-radius algorithm requires four point matrix products and is proven to compute an enclosure of the exact matrix product with a radius that is overestimated by a factor[1] at most 1.5.

In [13], Ogita and Oishi proposed faster midpoint-radius algorithms computing the interval matrix product in approximately twice the cost of a point matrix product. However, the known bound for the radius overestimation is worse than with the previous algorithm.

[1] Roundoff errors are neglected in radius overestimation factors given here. This is acceptable when the input intervals are large enough, say $\operatorname{rad} \boldsymbol{x} > 10^{-12} \operatorname{mid} \boldsymbol{x}$ for double precision.

In his PhD thesis [12, Chapter 2], Nguyen exhibited another interval matrix product algorithm that overestimates the radius by a factor less than 1.18 at a computational cost less than twice the initial midpoint-radius algorithm from Rump.

In [16], Rump improved the control of rounding errors in the product of midpoint matrices with the following bound:

Theorem 2.1 *Let $A \in \mathbb{F}^{m\times k}$ and $B \in \mathbb{F}^{k\times n}$ with $2(k+2)\mathtt{u} \leq 1$ be given, and let $C = fl_\square(A \cdot B)$ and $\Gamma = fl_\square(|A| \cdot |B|)$. Here C may be computed in any order, and we assume that Γ is computed in the same order. Then*

$$|fl_\square(A \cdot B) - A \cdot B| \leq fl_\square\left(\frac{k+2}{2}\mathrm{ulp}(\Gamma) + \frac{1}{2}\mathtt{u}^{-1}\eta\right)$$

where the inequality applies componentwise. In this theorem and in what follows, $\mathbb{F}$ denotes the set of floating-point numbers, $\mathbb{IF}$ the set of intervals with floating-point (midpoint-radius in our case) representation, $\mathtt{u}$ and η respectively the unit roundoff and the smallest positive normal floating-point number, $\mathrm{ulp}(\Gamma)$ the unit in the last place[2] of the components of Γ, and $fl_\square(E)$ (resp. $fl_\Delta(E)$) indicates that every operation in the arithmetic expression E is evaluated with rounding to nearest (resp. rounding towards $+\infty$). Using this bound, Rump transformed the above algorithms eliminating one point matrix product from his previous interval matrix product and two from Nguyen's. We reproduce here the improved algorithms `MMMul3` (Algorithm 1) and `MMMul5` (Algorithm[3] 2) without the additional conversions from and to the usual infimum-supremum representation of matrices.

Algorithm 1 `MMMul3`

Input: $\boldsymbol{A} = \langle M_{\boldsymbol{A}}, R_{\boldsymbol{A}}\rangle \in \mathbb{IF}^{m\times k}, \boldsymbol{B} = \langle M_{\boldsymbol{B}}, R_{\boldsymbol{B}}\rangle \in \mathbb{IF}^{k\times n}$
Output: $\boldsymbol{C}_3 \supseteq \boldsymbol{A} \cdot \boldsymbol{B}$
1: $M_{\boldsymbol{C}} \leftarrow fl_\square(M_{\boldsymbol{A}} \cdot M_{\boldsymbol{B}})$
2: $R'_{\boldsymbol{B}} \leftarrow fl_\Delta((k+2)\mathtt{u}|M_{\boldsymbol{B}}| + R_{\boldsymbol{B}})$
3: $R_{\boldsymbol{C}} \leftarrow fl_\Delta(|M_{\boldsymbol{A}}| \cdot R'_{\boldsymbol{B}} + R_{\boldsymbol{A}} \cdot (|M_{\boldsymbol{B}}| + R_{\boldsymbol{B}}) + \mathtt{u}^{-1}\eta)$
4: **return** $\langle M_{\boldsymbol{C}}, R_{\boldsymbol{C}}\rangle$

Algorithm 2 `MMMul5`

Input: $\boldsymbol{A} = \langle M_{\boldsymbol{A}}, R_{\boldsymbol{A}}\rangle \in \mathbb{IF}^{m\times k}, \boldsymbol{B} = \langle M_{\boldsymbol{B}}, R_{\boldsymbol{B}}\rangle \in \mathbb{IF}^{k\times n}$
Output: $\boldsymbol{C}_5 \supseteq \boldsymbol{A} \cdot \boldsymbol{B}$
1: $\rho_{\boldsymbol{A}} \leftarrow \mathrm{sign}(M_{\boldsymbol{A}}) .* \min(|M_{\boldsymbol{A}}|, R_{\boldsymbol{A}})$
2: $\rho_{\boldsymbol{B}} \leftarrow \mathrm{sign}(M_{\boldsymbol{B}}) .* \min(|M_{\boldsymbol{B}}|, R_{\boldsymbol{B}})$
3: $M_{\boldsymbol{C}} \leftarrow fl_\square\left(M_{\boldsymbol{A}} \cdot M_{\boldsymbol{B}} + \rho_{\boldsymbol{A}} \cdot \rho_{\boldsymbol{B}}\right)$
4: $\Gamma \leftarrow fl_\square\left(|M_{\boldsymbol{A}}| \cdot |M_{\boldsymbol{B}}| + |\rho_{\boldsymbol{A}}| \cdot |\rho_{\boldsymbol{B}}|\right)$
5: $\gamma \leftarrow fl_\Delta\left((k+1)\mathrm{ulp}(\Gamma) + \frac{1}{2}\mathtt{u}^{-1}\eta\right)$
6: $R_{\boldsymbol{C}} \leftarrow fl_\Delta\left((|M_{\boldsymbol{A}}| + R_{\boldsymbol{A}}) \cdot (|M_{\boldsymbol{B}}| + R_{\boldsymbol{B}}) - \Gamma + 2\gamma\right)$
7: **return** $\langle M_{\boldsymbol{C}}, R_{\boldsymbol{C}}\rangle$

[2] see [10, Section 2.6] for a definition.
[3] Operations $.*$ in lines 1 and 2 are the elementwise multiplication, as in Matlab notation.

Finally, Ozaki et al. in [14] used coarser but faster estimates for the radius of the product and proposed several algorithms for interval matrix multiplication at the cost of one, two and three calls to point matrix multiplication.

By relying on floating-point matrix-matrix multiplications, the previous algorithms were designed to be easy to implement efficiently. However, some care is needed and the next section discusses some implementation issues that can easily be overlooked.

3 Implementation Issues

The midpoint-radius representation of matrices allows to use calls to floating-point matrix products for the implementation of the interval matrix product. Many software libraries (for example GotoBLAS [3] or ATLAS [19]) offer optimized implementations of matrix operations following the interface defined by the Basic Linear Algebra Subprograms Technical Forum Standard [2]. Processor vendors also provide BLAS libraries with state-of-the-art implementation of matrix-matrix products for their own architectures, for instance Intel Math Kernel Library (MKL) and AMD Core Math Library (ACML) for x86.

Although BLAS matrix-matrix multiplication gives access to high-performance and portability, implementing `MMMul3` (Algorithm 1) or `MMMul5` (Algorithm 2) with them requires that two hypotheses are satisfied: first, that the rounding mode is taken into account by the library and second, that $M_{\boldsymbol{A}} \cdot M_{\boldsymbol{B}}$ and $|M_{\boldsymbol{A}}| \cdot |M_{\boldsymbol{B}}|$ are computed in the same order.

3.1 Rounding Modes

We mention here some situations where BLAS implementations fall short of the first hypothesis when the rounding mode in use is not the default rounding to nearest (see [8] for a thorough discussion). For instance, in order to calculate an overestimated result, a matrix-matrix product using a Strassen-like recursive algorithm [17] should compute overestimations for terms that are added and underestimations for terms that are subtracted but this would ruin its advantage over the classical algorithm. Implementations of extended precision BLAS [2, chapter 4] are another example. For matrix operations in particular, the reference implementation [9] uses double-double arithmetic which makes intensive use of error-free transformations [10, chapter 4] that require rounding to nearest mode.

Nonetheless, the radius of the interval product in Algorithms 1 and 2 above must be computed with rounding towards $+\infty$ to guarantee that the result accounts for all possible roundoff errors and contains the exact product.

3.2 Computation Order of $M_{\boldsymbol{A}} \cdot M_{\boldsymbol{B}}$ and $|M_{\boldsymbol{A}}| \cdot |M_{\boldsymbol{B}}|$

The second hypothesis comes from Theorem 2.1. The BLAS interface does not provide any function that computes simultaneously and in the same order a matrix-matrix product and the product of the absolute value of the inputs. So, the implementations of midpoint-radius algorithms have to compute the quantities of Theorem 2.1 with successive calls to floating-point matrix products, as in lines 3 and 4 of Algorithm 2. However, no order of operation is specified by the BLAS standard, so Theorem 2.1 may not be applicable. The orders of the floating-point operations performed by

the two multiplications are even more likely to differ when the BLAS library is itself multithreaded.

In that respect, it is useful to note that legal obligations and verification constraints of their users cause some vendors to address the problem of the reproducibility of numerical results between different processors or from run to run. For these reasons, the version 11.0 of MKL (see [18] or [6, Chapter 7]) now provides modes of execution where the user can control, at some loss in efficiency, the task scheduling and the type of computing kernels in use. In these modes, identical numerical results are guaranteed on different processors when they share the same architecture and run the same operating system. Moreover, reproducibility from run to run is ensured under the condition that, in all executions, the matrices have the same memory alignment and the number of threads remains constant.

We can use this kind of control to solve the problem of the computation order of $M_{\boldsymbol{A}} \cdot M_{\boldsymbol{B}}$ and $|M_{\boldsymbol{A}}| \cdot |M_{\boldsymbol{B}}|$. As the processor and the OS remain the same during the computation of the two products computations, it suffices, first, to compute $M_{\boldsymbol{A}} \cdot M_{\boldsymbol{B}}$, second, to transform in place matrix components into their absolute values ensuring the identity of memory alignments, then third, to recompute the product on the new input with the same number of threads. The drawback of this solution is its specificity to the Intel MKL as long as other libraries do not adopt the same control for numerical reproducibility.

In order to overcome these uncertainties on the execution of an arbitrary BLAS library, we present in the next section several implementations of Algorithm `MMMul5` that verify the two assumptions while still being efficient on a multicore target. In the following, Algorithm `MMMul5` has been preferred to Algorithm `MMMul3` because it is more computationally intensive, so the overhead of a correct implementation with respect to one that is linked against an optimized BLAS library is more apparent in the experimental measures. (Another, marginal, reason is that the bound of Theorem 2.1 does not appear directly in Algorithm `MMMul3`.)

4 Expressing Parallelism for a Multicore Target

In this section, we present several implementations of `MMMul5` in the C programming language for a multicore target.

The following few assumptions will be made on the structure of the interval matrices in midpoint-radius representation. First, interval matrices are manipulated as pairs of arrays in the row-major order that is usual in C. Second, the array of midpoints and the array of radii are assumed to share the same alignment and stride. The stride of the pair may be different from the column dimension as this allows to handle submatrices, but no padding will be made to align rows with a given value.

The C-99 standard defines accesses to the floating-point environment and in particular to rounding modes through the `fegetround` and `fesetround` functions which allow readings and changes of rounding modes in a portable manner. However, even up-to-date versions of some compilers do not take into account the changes of rounding modes in their optimization phases (cf. the long-running bug of GCC [1]), yielding possibly incorrect results when a call to `fesetround` is crossed by a floating-point operation. Calling directly an assembly instruction to change the rounding mode solves this problem, but this workaround is platform-dependent. Nevertheless, in the implementations of `MMMul5` below, we use the portable C-99 functions just before a loop or a call to a BLAS library function as we did not observe the bug mentioned above in

these conditions.

Beside correctness problems, we aim to provide an implementation that is efficient on a multicore target. Two main features of the multiplication of dense matrices make them very well-suited for parallelization. First, they offer a high degree of data independence, as each component of the product could virtually be computed regardless of the others. This offers opportunity for calculating sub-matrix blocks in different threads. Second, the property of density of the matrices permits regularity and contiguity of memory accesses. This last condition is necessary to benefit from the memory cache system of processors.

We exploit below the coarse-grained parallelism of the block calculation at the thread level by using OpenMP constructs. Furthermore, we translate the fine-grained parallelism of contiguous data accesses and similar computations into instruction level parallelism through vector instructions.

Ideally, this last step should be handled by the compiler auto-vectorization, but, as discussed below in Section (4.4), that is beyond current compiler capabilities.

4.1 Version with a Parallel BLAS Library

It has been shown in Section 2 that the algorithms for interval matrix products using the midpoint-radius representation rely on floating-point matrix products. As advocated by their authors, this approach is a straightforward means to benefit from optimized BLAS libraries at a small development cost. From this point of view, `MMMul5` can be parallelized on a multicore machine by using a multithreaded BLAS library.

We present below a possible implementation of the first part in rounding to nearest (lines 1 to 4) of Algorithm 2. In the BLAS implementation of Algorithm 3, lines 1, 4, 7, and 10 translate directly to calls to the `xgemm` function, which takes two scalars α and β and three matrices A, B, and C as parameters and computes $C \leftarrow \alpha A \cdot B + \beta C$.

Algorithm 3 Rounding to nearest part of `MMMul5BLAS`

1: $M_{\boldsymbol{C}} \leftarrow 1 M_{\boldsymbol{A}} \cdot M_{\boldsymbol{B}} + 0 M_{\boldsymbol{C}}$
2: $T_1 \leftarrow |M_{\boldsymbol{A}}|$
3: $T_2 \leftarrow |M_{\boldsymbol{B}}|$
4: $\Gamma \leftarrow 1 T_1 \cdot T_2 + 0\Gamma$
5: $T_3 \leftarrow \min(T_1, R_{\boldsymbol{A}})$
6: $T_4 \leftarrow \min(T_2, R_{\boldsymbol{B}})$
7: $\Gamma \leftarrow 1 T_3 \cdot T_4 + 1\Gamma$
8: $T_5 \leftarrow \text{sign}(M_{\boldsymbol{A}}) .* T_3$
9: $T_6 \leftarrow \text{sign}(M_{\boldsymbol{B}}) .* T_4$
10: $M_{\boldsymbol{C}} \leftarrow 1 T_5 \cdot T_6 + 1 M_{\boldsymbol{C}}$

Alas, as discussed in Section 3.2, we cannot guarantee that this implementation does always satisfy the Theorem 2.1 hypothesis on the order of computation.

4.2 Version with a Parallel Loop for the Rounding-to-Nearest Part

The simplest means of preserving the order of operations between products and products of absolute value is to combine their computations in the same loop. We call

`MMMul5frn` (for *fused rounding to nearest loop*) the Algorithm 4 below that computes simultaneously $M_{\boldsymbol{C}}$ and Γ in the classical three nested loops of a matrix-matrix product.

Algorithm 4 Round to nearest part of `MMMul5frn`

1: **for** $i \leftarrow 1, m$ **do** {OpenMP `parallel for`}
2: **for** $l \leftarrow 1, k$ **do**
3: **for** $j \leftarrow 1, n$ **do**
4: $a \leftarrow M_{\boldsymbol{A}il}; c \leftarrow R_{\boldsymbol{A}il}$
5: $b \leftarrow M_{\boldsymbol{B}lj}; d \leftarrow R_{\boldsymbol{B}lj}$
6: $e \leftarrow \text{sign}(a)\min(|a|, c)$
7: $f \leftarrow \text{sign}(b)\min(|b|, d)$
8: $p \leftarrow ab + ef$
9: $M_{\boldsymbol{C}ij} \leftarrow M_{\boldsymbol{C}ij} + p$
10: $\Gamma_{ij} \leftarrow \Gamma_{ij} + |p|$
11: **end for**
12: **end for**
13: **end for**

The choice of the order of the three loops is important with respect to the locality of the memory accesses. In `MMMul5frn`, the inner loop spans the columns of $M_{\boldsymbol{C}}$ and Γ, so that memory is written contiguously. This facilitates hardware prefetching as well as vectorization. The external loop traverses the rows of $M_{\boldsymbol{C}}$ and Γ and is annotated with an OpenMP `parallel for` directive to distribute blocks of rows to multiple threads. This choice of indices for the external and inner loops also avoids false sharing[4] between cache lines of different cores when the matrices are stored in row-major order.

Algorithm `MMMul5frn` uses less memory bandwidth than the BLAS implementation described in Section 4.1 by saving temporary memory buffers, each component of ρ values (lines 1 and 2 of Algorithm 2) and of absolute values of $M_{\boldsymbol{A}}$ and $M_{\boldsymbol{B}}$ being computed on-the-fly.

4.3 Version with a Blocked Parallel Loop for the Rounding-to-Nearest Part

When the matrix $\boldsymbol{C}$ is large enough, a row vector of $M_{\boldsymbol{C}}$ and a row vector of Γ cannot be kept entirely and simultaneously in the first level cache. As the inner loop accumulates partial products in $M_{\boldsymbol{C}}$ and Γ, the first components are evicted from the cache when the last ones are computed. At the next iteration, the foremost ones are needed again and in turn evict the end ones. With the previous choice of loop nesting, this leads to one cache miss per component at each iteration over the common dimension of $M_{\boldsymbol{A}}$ and $M_{\boldsymbol{B}}$.

[4] The first levels of memory caches are usually private to the core. When two private caches try to maintain a coherent vision of the same memory region, they use a system of monitoring and copy. Modifying a data in some memory block that is duplicated in another cache triggers this costly mechanism, even if the other core does not use the modified data but a nearby one contained in the same cache line. This phenomenon is called *false sharing* (cf. [5, Section 4.3]).

To avoid the cache misses of `MMMul5frn`, we propose a blocked implementation `MMMul5bfrn` where the matrices $M_{\boldsymbol{C}}$ and Γ are split into blocks of rows that can be stored simultaneously in the same cache. If the capacity of the cache is too small to contain two complete rows, then the rows of $M_{\boldsymbol{C}}$ and Γ to be computed are also divided into pieces that are small enough to fit in the cache at the same time.

The `MMMul5frn` and `MMMul5bfrn` versions only differ in the blocking of m and n dimensions, the outer loop is distributed to threads with the OpenMP `parallel for` construct in both implementations and the inner loop block (lines 4 to 10 in Algorithm 4) remains unchanged.

4.4 Version with an Explicitly Vectorized Kernel

The inner loop block of Algorithm 4 can be very easily and efficiently translated into a sequence of vector instructions without any conditional branch. In fact, signs of floating-point numbers can be extracted and copied with bit-masks and logical operations, while the minimum of two floating-point values is usually provided either as a vector instruction (e.g. `FMIN` on Sparc), or it can be implemented as a combination of the comparison instruction that issues bit masks of all ones or all zeros and logical operations between bit masks and floating-point data (e.g. with AltiVec and SSE instruction sets).

Unfortunately, the transformations that convert the inner loop into vectorized code are still too complicated to be handled by current compilers. Thus, for the purpose of comparing execution time of optimized computation kernel in Section 5, we manually translated the inner loop block Algorithm `MMMul5bfrn` into SSE2 code and we denote by `MMMul5bfrn-sse2` the new implementation. As expected, the gain in execution time is important (see Section 5.2), but the code is no more portable.

4.5 Rounding-Upwards Part

The computation of $R_{\boldsymbol{C}}$ uses the rounding to $+\infty$ mode, as shown in the lines 5 and 6 of Algorithm 2. It involves one matrix-matrix product and we cannot ensure that an `xgemm` function of a given BLAS library always yields the expected overestimation (see Section 3.1). Therefore, if we want to guarantee the inclusion of the exact product in the computed product, an implementation of Algorithm 2 has to be statically linked against a trusted BLAS library if it uses the BLAS `xgemm` function. Another solution is to write the rounding-upward part with the classical three loops algorithm.

In the latter case, the implementation can use strategies similar to the ones used above for the rounding-to-nearest part. The computation of lines 5 and 6 of Algorithm 2 can be mixed into the same inner loop, saving some accesses to temporary memory buffers; remaining memory operations can be organized so that memory is written in a contiguous manner; and the external loop can be easily parallelized by distributing iterations to OpenMP threads.

5 Performance Results

We now present experimental measures and comparisons of the execution time for the implementations presented in the previous section, namely: `MMMul5` BLAS (4.1), `MMMul5frn` (4.2), `MMMul5bfrn` (4.3), and `MMMul5bfrn-sse2` (4.4). In order to evaluate the costs of the different implementations of the rounding-to-nearest part, all the tested

computation kernels share the same rounding-upwards part implemented with calls to the `xgemm` function.

In the following measures, all floating-point numbers are in double precision (i.e. IEEE-754 `Binary64` type).

5.1 Experimental Setup

We experiment with a multiprocessor system composed of 4 eight-cores Xeon E5-4620 (Sandy Bridge) processors. Operating system and software being used are described in Table 1. In particular, the BLAS implementation is linked with the available MKL version 10.3, which does not provide the new modes of execution ensuring the reproducibility of the numerical results discussed in Section 3.2.

All computation kernels are compiled with GCC compiler version 4.7.2 and the following compiler options: `-O2` sets the level of optimization, `-m64` is required when linking with the 64-bit version of the MKL, `-frounding-math` disable floating-point optimizations that are valid only with the default rounding-to-nearest mode, `-mfpmath=sse -msse2` enable the SSE extensions, and `-ftree-vectorize` unconditionally triggers the loop vectorization phase. Figures do not differ noticeably with the `-O3` level of optimization, except for very small matrices.

	4 processors Intel Xeon E5-4620
number of cores	$4 \times 8 = 32$
operating system	Linux version 3.2.0 (Debian Wheezy)
compiler suite	gcc 4.7.2
BLAS library	Intel MKL version 10.3.4
OpenMP library	Gnu OpenMP library

Table 1: Description of the system used for performance results.

The processor characteristics are detailed in Table 2. The Hyper-Threading and Turbo Boost capabilities are disabled for stable and reproducible measurements.

	Intel Xeon E5-4620
clock speed	2.20GHz
number of cores	8
SIMD instruction set	SSE2, AVX
L1 data caches	32KB per core, 8-ways
L2 caches	256KB per core, 8-ways
L3 cache	16MB shared, 16-ways

Table 2: Description of Xeon E5-4620 processor.

As discussed in Section 4, the three kernel versions `MMMul5frn`, `MMMul5bfrn`, and `MMMul5bfrn-sse2` are multithreaded using OpenMP. OpenMP libraries are controlled by environment variables that are also taken into account by the MKL library for its own multithreading management. In our experiments, we set the following environment variables: `OMP_PROC_BIND` is set to `true` in order to avoid thread migration

from one core to another and the associated time overhead, and `OMP_DYNAMIC` is set to `false` so as to benefit from all the available cores when wanted without being under the control of dynamic adjustment of the number of threads.

5.2 Execution Time with 32 Threads

We first evaluate and compare the execution time of the given computation kernels with 32 concurrent threads. The measured timings are displayed on Figure 1.

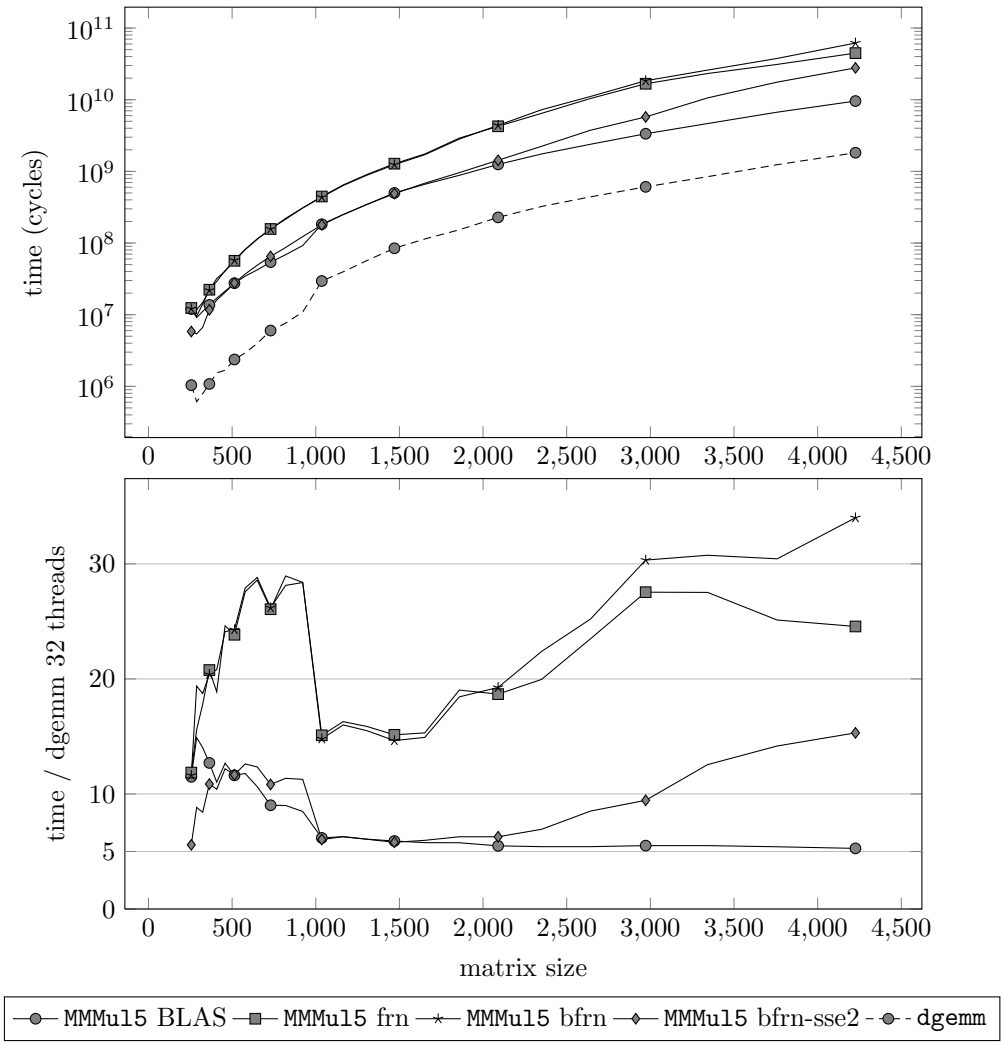

Figure 1: Execution time – 32 threads.

The graph at the top of Figure 1 represents the execution time of the interval

matrix-matrix products discussed in Section 4 along with the floating-point matrix-matrix product (`dgemm` function) for input pairs of square matrices from dimension 200 to 4,200. It can be noticed from these measures that `dgemm` and `MMMul5` BLAS are subject to a sudden increase of execution time around matrix dimension of 1,000. We impute this phenomenon to the capacity of the L3 cache that can contain simultaneously only two square matrices of dimension 1,024. With matrices of dimension more than 1,000, the slope of the MKL timings tends to be more and more gentle, demonstrating the optimization of the library for large matrices. Kernels `MMMul5frn` and `MMMul5bfrn` present comparable timings until dimension 2,000. For larger matrices, the blocked version `MMMul5bfrn` appears to be the slowest one. Finally, this set of measures proves that a blocked implementation correctly translated in vector instructions as `MMMul5bfrn-sse2` can compete with the BLAS implementation for matrices of dimension less than 2,000.

In order to exhibit the slowdown of interval compared to floating-point matrix products, the ratio of execution time of the different implementations to the time of a `dgemm` execution are displayed at the bottom of Figure 1. The following remarks can be made about these results. As the dimension of the input matrices increases, the five `dgemm` calls dominate other operations (absolute values, minimum selections, and sign copies, see Algorithm 3), and the ratio perceptibly tends to 5. Except for very small matrices, `MMMul5frn` and `MMMul5bfrn` implementations are much less efficient than the BLAS implementation. The sudden decrease of the ratio value around dimension 1,000 is to be related to the time increase of the reference `dgemm` already discussed. The explicit SSE2 version `MMMul5bfrn-sse2` is about twice as fast as the `MMMul5bfrn`, demonstrating that the compiler is unable to efficiently auto-vectorize the code. It is anticipated that another 2-fold factor could be reached with a version exploiting the AVX instruction set where the vector width is of four double precision numbers, twice the width of SSE vectors. However, we observe little gain provided by the `-mavx` compiler option, as the compiler is unable to vectorize the inner loop.

To compare the best implementations, `MMMul5bfrn-sse2` is faster than the distrusted `MMMul5` BLAS implementation for matrices up to the dimension 500, then `MMMul5bfrn-sse2` gets a bit slower (about 12 times a `dgemm` execution) until dimension 1,000, and from this last dimension up to 2,000, both have similar performance. For matrix dimensions over 2,000, `MMMul5bfrn-sse2` gets slower and slower, this is due to the fact that we have limited here the blocking strategy to the first level of cache. An improved version with several levels of blocking and thresholds tuned with the cache properties of the underlying architecture should remain competitive with the BLAS implementation for larger matrices.

5.3 Scalability

We now analyze the behaviors of the computation kernels when the number of threads rises while the matrix dimension remains constant: these behaviors relate to what will be called below the *strong scalability*.

Figure 2 represents the execution time measures of four algorithms: (a) at the top-left corner, the floating-point matrix-matrix product of the BLAS library, (b) at the top-right, the `MMMul5` BLAS implementation (see Section 4.1), (c) bottom-left, the blocked `MMMul5bfrn` version presented in Section 4.3, and (d) bottom-right, the blocked `MMMul5bfrn-sse2` version that has been manually vectorized (discussed in Section 4.4). Each sub-figure presents four groups of measurement for square matrix of dimensions 512, 1,024, 2,048 and 4,096 respectively. Each group is divided into six bars related

to the number of threads used for the run (from 1 thread to 32 threads). Ordinates represent the execution time ratio of each algorithm compared to the execution time of the `dgemm` function for the same dimension of matrix with the same number of threads. Hence, every leftmost bar of each group of sub-figure (a) corresponds to a value of 1.

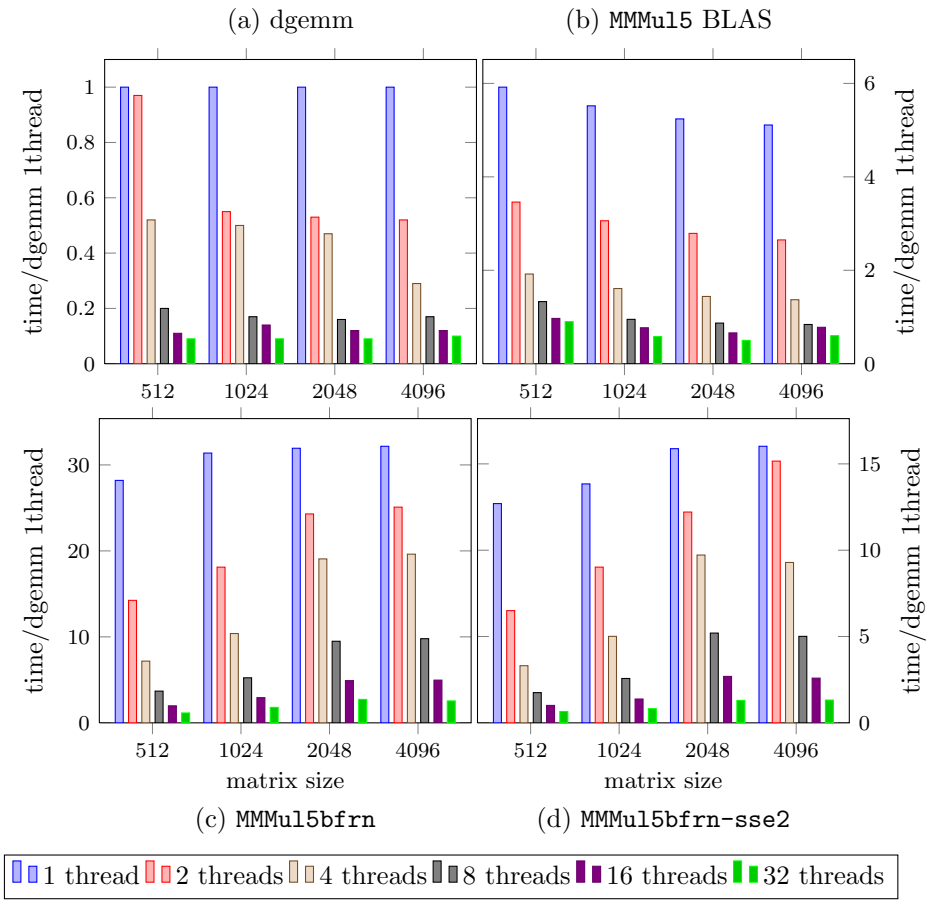

Figure 2: Scalability.

The behavior of the `dgemm` function is not regular. For instance, there is little gain in using two threads for multiplying two 512×512 matrices compared to the execution time with one thread. By contrast, the gain between 4 threads and 8 threads is more than twofold. The behavior of the `MMMul5` BLAS implementation is more regular but the gain factor tends to decrease when more than 8 threads are employed. The strong scalability of `MMMul5bfrn` is close to ideal (a doubling of the thread number causes a halving of the run time) for small matrices. Timings for `MMMul5bfrn-sse2` are half of the corresponding ones for `MMMul5bfrn`, this is due to the vector size, as explained before. Therefore, the patterns of Figure (d) are the same as Figure (c), reduced by a factor of one half.

6 Conclusion

The new results presented in Section 4 are algorithms for interval matrix multiplication that are really guaranteed to contain the exact result and that are efficient on a multicore architecture, as seen in Section 5.
We have seen that implementing interval algorithms on high-performance architectures amplifies the recurrent question of the efficiency of these algorithms, as poor performance even degrades compared to the performance of (non-guaranteed but fast) floating-point algorithms. However, getting reliability and performance is even more difficult than on sequential architectures, as many issues are obscured by lack of specifications (for instance, what happens to the rounding mode in a multithreaded environment) or by specific issues, such as the lack of reproducibility of the numerical result from run to run.
From the implementation developed in this paper, we may draw the following methodological conclusions. Firstly, develop interval algorithms that are based on well-established numerical bricks (such as the `gemm` of BLAS in this work), so as to benefit from their optimized implementation. A second step could be to convince developers and vendors of these bricks to clearly specify their behavior, in particular with regards to rounding mode. However, we have observed that this may not suffice, as this was the case with the issue on the order in which floating-point operations should be performed at each call. A third step is thus to replicate the work done for the optimization of the considered numerical bricks, and to adapt it to the specificities and requirements of the interval algorithm, as it has been done to compute "simultaneously" $A \cdot B$ and $|A| \cdot |B|$. For the interval matrix multiplication in particular, it would be worth developing an Atlas-like autotuning of the blocked version in order to optimize the usage of all cache levels and not only of the cache of level 1, as it had been done here. This would prevent the loss of performance when the matrix size increases too much.
However, following such a methodology requires that the programmer gathers skills both in interval arithmetic and in HPC programming and this is rare.
This methodology could lead to the development of Interval-BLAS, offering various algorithms with different accuracies (at least theoretically established) and good efficiency.

References

[1] Bug 34678 – optimization generates incorrect code with -frounding-math option. `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=34678`, January 2008.

[2] S. Blackford, G. Corliss, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, C. Hu, W. Kahan, L. Kaufmann, B. Kearfott, F. Krogh, X. Li, Z. Maany, A. Petitet, R. Pozo, K. Remington, W. Walster, C. Whaley, J. Wolff von Gudenberg, and A. Lumsdaine. *Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard*, 2001. `http://www.netlib.org/blast-forum/`.

[3] Kazushige Goto and Robert van de Geijn. High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software*, 35(1):4:1–4:14, 2008.

[4] Y. He and C. H. Q. Ding. Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications. *The Journal of Supercomputing*, 18:259–277, 2001.

[5] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (4th ed.)*. Morgan Kaufmann, 2007.

[6] Intel. *Intel Math Kernel Library for Linux OS User's Guide*. `http://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/mkl_userguide_lnx/mkl_userguide_lnx.pdf`.

[7] R. B. Kearfott, M. T. Nakao, A. Neumaier, S. M. Rump, S. P. Shary, and P. van Hentenryck. Standardized notation in interval analysis. *Reliable Computing*, 15(1):7–13, 2010.

[8] Christoph Lauter and Valérie Ménissier-Morain. There's no reliable computing without reliable access to rounding modes. In *SCAN 2012 Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics*, pages 99–100, 2012.

[9] Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung, and Daniel J. Yoo. Design, Implementation and Testing of Extended and Mixed Precision BLAS. *ACM Transactions on Mathematical Software*, 28(2):152–205, June 2002.

[10] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[11] A. Neumaier. *Interval methods for systems of equations*. Cambridge University Press, 1990.

[12] Hong Diep Nguyen. *Efficient algorithms for verified scientific computing: numerical linear algebra using interval arithmetic*. PhD thesis, École Normale Supérieure de Lyon - Université de Lyon, 2011. `http://hal-ens-lyon.archives-ouvertes.fr/ensl-00560188`.

[13] Takeshi Ogita and Shin'ichi Oishi. Fast inclusion of interval matrix multiplication. *Reliable Computing*, 11(3):191–205, 2005.

[14] Katsuhisa Ozaki, Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Fast algorithms for floating-point interval matrix multiplication. *Journal of Computational and Applied Mathematics*, 236:1795–1814, 2012.

[15] Siegfried M. Rump. Fast and parallel interval arithmetic. *BIT*, 39:534–554, 1999.

[16] Siegfried M. Rump. Fast interval matrix multiplication. *Numerical Algorithms*, 61(1):1–34, 2012.

[17] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[18] R. Todd. Introduction to Conditional Numerical Reproducibility (CNR). `http://software.intel.com/en-us/articles/introduction-to-the-conditional-numerical-reproducibility-cnr`, 2012.

[19] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. In *Conference on High Performance Networking and Computing*, pages 1–27. IEEE Computer Society, 1998.