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 using optimized floating-point BLAS and, consequently, the performance benefit from the performance of the BLAS routines.

Several variants of interval matrix multiplication have been proposed, which correspond to various trade-offs between accuracy and efficiency, including some efficient ones proposed by Rump in 2012. However, 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 necessarily satisfied by most implementations of the BLAS.

In this paper, an algorithm for interval matrix product is proposed that satisfies this assumption. Furthermore, several optimization procedures are proposed, and the implementation on a multicore architecture

*Submitted: February 21, 2013; Revised: November 22, 2013; Accepted: November 26, 2013.

compares reasonably well with a non-guaranteed implementation based on MKL, the optimized BLAS of Intel: the overhead is less than 2 for matrix size up to 3,500. This implementation also exhibits 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 this work particularly, 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 changing the rounding mode, 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 is 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 floating-point addition is not associative, and thus 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 difficulty is related to the programming of multicore architectures. Since optimized libraries, such as MKL [6], optimized by Intel for its processors, exhibit good performance, a first idea is to try to use these libraries to benefit from their performance. However, since 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 performance 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 the faster ones in [15] that resort to 4 calls to BLAS floating-point matrix multiplication, and 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. 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 performance. Autotuned versions, such as ATLAS [19], provide good performance on various hardware by automatically tuning cache usage via block size, etc.: these 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. This architecture still relies on a shared memory: going to distributed architecture is a major step we did not take, since issues become fundamentally different from shared-memory or sequential ones.

We review the algorithms from [16] and the assumption on which they rely in Section 2. Since this assumption is no longer assured to be true with the implementation of the BLAS on the architecture we consider, as explained in Section 3, we give up the use of the BLAS. The main result of this paper is **the implementation of dense interval matrix multiplication which is fully guaranteed**, as the requirement from [16] is fulfilled, **and which gives good performance**, as optimization by hand is done where needed. The details are elaborated in Section 4. In particular, exploiting cache usage through a version by blocks yields performance which is 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 coefficients that are points: 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 $[\underline{A}, \overline{A}]$; this 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 enables faster computer implementations of matrix operations 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 the 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.

[1] Roundoff errors are neglected in the 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 [13], Ogita and Oishi proposed faster midpoint-radius algorithms that compute 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.

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 (the inequality applies componentwise):

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*

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

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 $+\infty$) mode. 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** $\langle M_{\boldsymbol{C}}, R_{\boldsymbol{C}}\rangle$

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 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.

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

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: $\rho_{\boldsymbol{A}} \leftarrow \text{sign}(M_{\boldsymbol{A}}) .* \min(|M_{\boldsymbol{A}}|, R_{\boldsymbol{A}})$
2: $\rho_{\boldsymbol{B}} \leftarrow \text{sign}(M_{\boldsymbol{B}}) .* \min(|M_{\boldsymbol{B}}|, R_{\boldsymbol{B}})$
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: $\gamma \leftarrow fl_{\Delta}\left((k+1)\text{ulp}(\Gamma) + \frac{1}{2}\mathtt{u}^{-1}\eta\right)$
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** $\langle M_{\boldsymbol{C}}, R_{\boldsymbol{C}} \rangle$

3 Implementation Issues

The midpoint-radius representation of matrices allows one to resort to floating-point matrix products for the implementation of interval matrix products. 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 products for their own architectures, like, for instance, the Intel Math Kernel Library (MKL) and the AMD Core Math Library. Thus, BLAS libraries give access to high-performance and portability.

However, `MMMul3` (Algorithm 1) and `MMMul5` (Algorithm 2) require that two hypotheses are satisfied: first, the rounding mode is taken into account by the library and, second, $M_{\boldsymbol{A}} \cdot M_{\boldsymbol{B}}$ and $|M_{\boldsymbol{A}}| \cdot |M_{\boldsymbol{B}}|$ are computed in the same order (along with $\rho_{\boldsymbol{A}} \cdot \rho_{\boldsymbol{B}}$ and $|\rho_{\boldsymbol{A}}| \cdot |\rho_{\boldsymbol{B}}|$ in `MMMul5`).

3.1 Rounding Modes

We mention here some situations where implementations with BLAS 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, to calculate an overestimated result, a 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. Extended precision BLAS [2, Chapter 4] are another example. For matrix operations especially, 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 calculates a matrix product and the product of the absolute values of the inputs simultaneously and in the same order. Thus, 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 [2], 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, version 11.0 of the 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 $|M_{\boldsymbol{A}}| \cdot |M_{\boldsymbol{B}}|$. Since the processor and the OS remain the same during the computation of the two products, 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.

To overcome these uncertainties in the execution of an arbitrary BLAS library, we present several implementations of Algorithm `MMMul5` that verify the two assumptions while still being efficient on a multicore target in the next section. 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 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 row-major order[4], which 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 us 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, 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, yielding possibly incorrect results when they swap

[4]Note that, according to our experience, the `dgemm` function in MKL seems to be more efficient when using the column-major order.

a call to `fesetround` and a floating-point operation (cf. the long-running bug in GCC [1]). Calling an assembly instruction to change the rounding mode directly 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 loops, as we did not observe the bug mentioned above under 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 opportunities for calculating sub-matrix blocks in different threads. Second, 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.

Below, we exploit 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 these algorithms' authors, this approach is a straightforward means to benefit from optimized BLAS libraries for 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 mode (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 `gemm` 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 `MMMul5-BLAS`

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

Alas, as discussed in Section 3.2, lines 1 and 4 compute $M_{\boldsymbol{A}} \cdot M_{\boldsymbol{B}}$ and $|M_{\boldsymbol{A}}| \cdot |M_{\boldsymbol{B}}|$

with arithmetic operations in arbitrary orders. Therefore, we cannot guarantee that this implementation always satisfies the Theorem 2.1 hypothesis of identical orders.

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 refer to Algorithm 4 below as `MMMul5frn` (for *fused loops in rounding-to-nearest*); it simultaneously computes $M_{\boldsymbol{C}}$ and Γ in the classical three nested loops of a matrix product. At lines 8 and 10, note that ef is of the same sign as ab, so $|ab|+|ef| = |ab+ef| = |p|$.

Algorithm 4 Rounding-to-nearest part of `MMMul5frn`

Input: $\langle M_{\boldsymbol{A}}, R_{\boldsymbol{A}}\rangle$ and $\langle M_{\boldsymbol{B}}, R_{\boldsymbol{B}}\rangle$
Output: $M_{\boldsymbol{C}}$ and Γ accumulated in the *same order*

```
 1: for i ← 1, m do {OpenMP parallel for}
 2:   for l ← 1, k do
 3:     for j ← 1, n do
 4:       a ← M_A_il, c ← R_A_il
 5:       b ← M_B_lj, d ← R_B_lj
 6:       e ← sign(a) min(|a|, c)
 7:       f ← sign(b) min(|b|, d)
 8:       p ← ab + ef
 9:       M_C_ij ← M_C_ij + p
10:       Γ_ij ← Γ_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[5] between cache lines of different cores when the matrices are stored in row-major order.

Algorithm `MMMul5frn` uses less memory than the BLAS implementation described in Section 4.1, 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.

[5]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 datum 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]).

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 first ones are needed again and in turn evict the last 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 $M_{\boldsymbol{B}}$.

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 simultaneously fit into the cache.

The `MMMul5frn` and `MMMul5bfrn` versions differ only 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 (lines 4 to 10 in Algorithm 4) remains unchanged.

4.4 Version with an Explicitly Vectorized Kernel

The inner loop 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 these bit masks and the floating-point data (e.g. with AltiVec, SSE, and AVX 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 the optimized computation kernel in Section 5, we manually translated the inner loop block Algorithm `MMMul5bfrn` into AVX code; we call the new implementation `MMMul5bfrn-avx`. As expected, the gain in execution time is significant (see Section 5.3), but the code is no longer portable.

4.5 Rounding-Upwards Part

The computation of $R_{\boldsymbol{C}}$ uses the rounding to $+\infty$ mode, as shown in lines 5 and 6 of Algorithm 2. It involves one matrix product, and we cannot ensure that a `gemm` function of a given BLAS library always yields the expected overestimation (see Section 3.1). Therefore, if we want to guarantee 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 `gemm` 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; the external loop can be easily parallelized by distributing iterations to OpenMP threads.

5 Performance Results

We now present experimental measures and comparisons of execution times for the implementations presented in the previous section, namely: the non-guaranteed `MMMul5-BLAS` (4.1) and the guaranteed versions `MMMul5frn` (4.2), `MMMul5bfrn` (4.3), and `MMMul5bfrn-avx` (4.4). 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. This part is not implemented as described in Section 4.5, but simply with a BLAS matrix product (i.e. one call to the `dgemm` function).

5.1 Experimental Setup

We experiment with a shared memory system of 4 eight-core Xeon E5-4620 processors. To assure stable and reproducible measurement, the Hyper-Threading and Turbo Boost capabilities are disabled and the clock frequency is statically set to its highest possible value. The processor characteristics, operating system, and software being used are described in Table 1.

4 processors Intel Xeon E5-4620 (Sandy Bridge)	clock speed	2.20GHz
	SIMD instruction set	SSE2, AVX
	level 1 data caches	32KB per core
	level 2 caches	256KB per core
	level 3 cache	16MB, shared
number of cores	32 (4 processors × 8 cores)	
operating system	Linux version 3.2.0 (Debian Wheezy)	
compiler suite	GCC 4.7.2	
BLAS library	Intel MKL version 11.0.2	
OpenMP library	Gnu OpenMP library	

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

We compile with GCC 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` disables floating-point optimizations that are valid only with the default rounding-to-nearest mode, `-mfpmath=sse` imposes the use of SSE/AVX floating-point unit, `-mavx` enables the AVX-vector instruction set, and `-ftree-vectorize` unconditionally triggers the loop vectorization phase. Results do not differ noticeably with the `-O3` level of optimization, except for very small matrices.

As it has been discussed in Section 4, the three kernels `MMMul5frn`, `MMMul5bfrn`, and `MMMul5bfrn-avx` 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` to avoid thread migration from one core to another and the associated time overhead, and `OMP_DYNAMIC` is set to `false` to benefit

from all the available cores when wanted without being under the control of dynamic adjustment of the number of threads. Finally, we choose double precision (i.e. the IEEE-754 `Binary64` type) as the working precision in all of the following results.

5.2 Sequential Execution Time

We first evaluate and compare the sequential execution time of the computation kernels presented in Section 4. The measured timings are displayed on Figure 1.

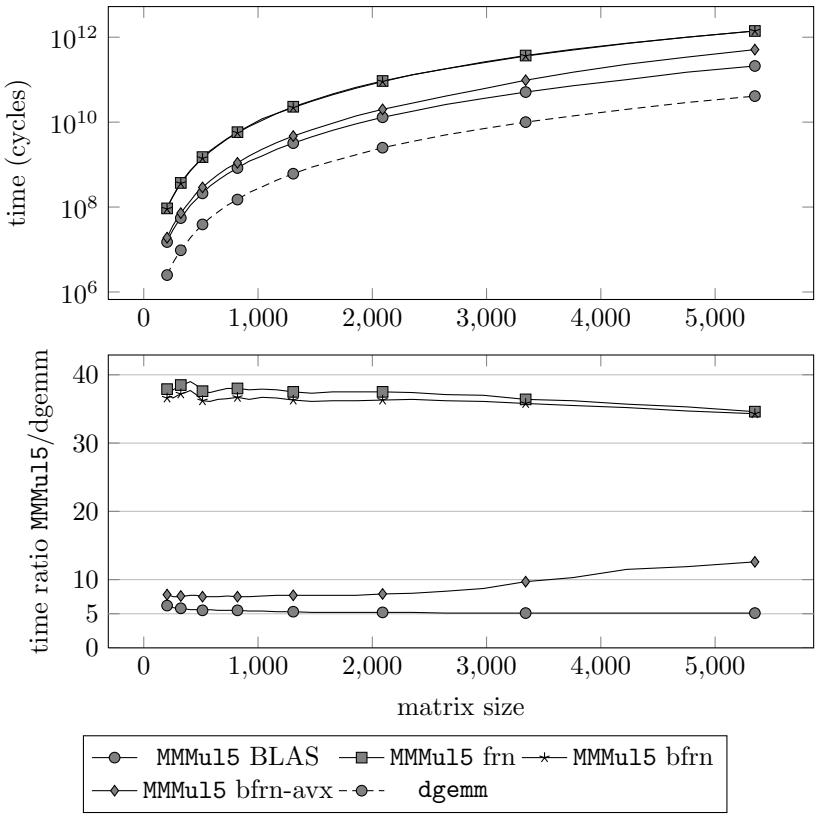

Figure 1: Execution time – Sequential.

The graph at the top of Figure 1 represents the execution time of the interval matrix products along with the floating-point matrix product (`dgemm`). The input is a pair of square matrices from dimension 200 to 5,350. To exhibit the slowdown of interval compared to floating-point matrix products, the ratios 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 authors of Algorithm 2 expected, the cost of `MMMul5-BLAS` is very close to 5 times the cost of a call to `dgemm`. Other implementations do not benefit from optimization of the MKL and

the `MMMul5frn` is about 35 times slower than `dgemm` algorithm. The blocked version `MMMul5bfrn` is slightly faster than `MMMul5frn` because of a better locality of memory accesses as explained in Section 4.3. The explicit AVX version `MMMul5bfrn-avx` is about four times as fast as `MMMul5bfrn`, demonstrating that the compiler is unable to efficiently vectorize the code by itself. Indeed, on the `MMMul5bfrn` code, we observed little gain provided by the `-mavx` compiler option, because GCC is unable to vectorize the inner loop. For small matrices (up to $3,500 \times 3,500$), `MMMul5bfrn-avx` is less than 2 times slower than the non-guaranteed `MMMul5-BLAS`.

5.3 Execution Time with 32 Threads

We now measure the execution time of the computation kernels when using one thread per core. The measured timings are displayed on Figure 2 in the placement described in the previous section.

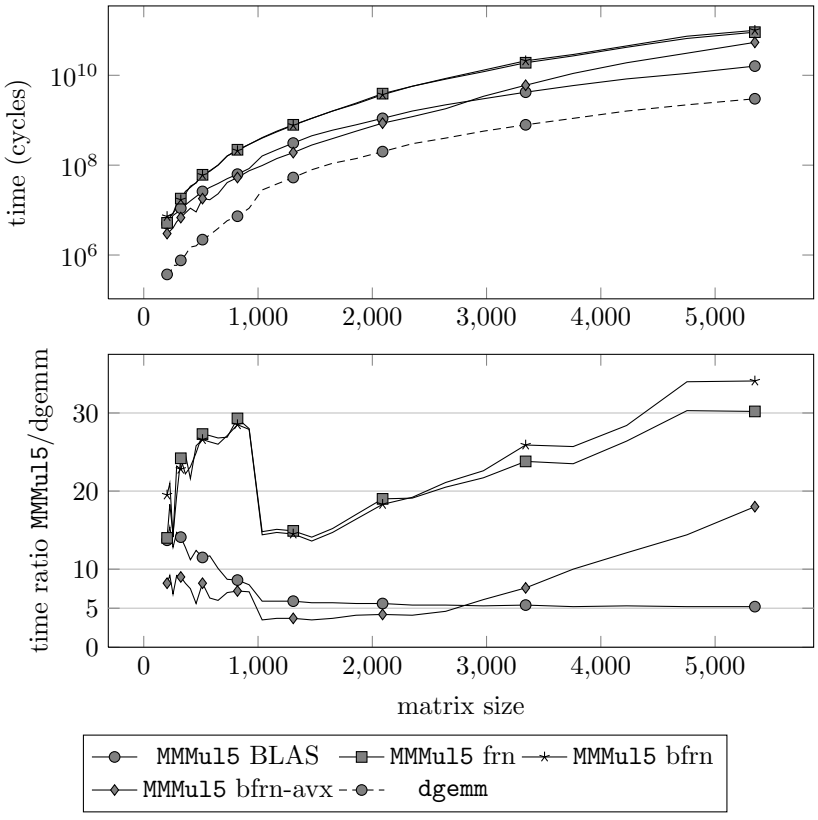

Figure 2: Execution time – 32 threads.

It can be noticed in the graph at the top of Figure 2 that `dgemm` and `MMMul5-BLAS` are subject to a sudden increase in execution time around matrix dimension 1,000. We ascribe this phenomenon to the capacity of the level 3 cache that can contain

simultaneously no more than two square matrices of dimension 1,024. With matrices of dimension larger 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 2,500. From dimension 500 to dimension 2,000, `MMMul5bfrn` undergoes less cache misses when accessing the first level of memory cache and is slightly more efficient than `MMMul5frn`. For larger matrices, the blocked version `MMMul5bfrn` appears to be the slowest one. This may be due to the greater management overhead of a higher number of OpenMP threads. Finally, this set of measures demonstrates that a blocked implementation correctly translated in vector instructions like `MMMul5bfrn-avx` can compete with the BLAS implementation for matrices of dimension less than 2,500.

Compared to the sequential execution, the relative behaviors of the kernels are markedly different, as it can be noticed from the graph at the bottom of Figure 2. For small input, the overhead of non-BLAS operations (absolute values, minimum selections, and sign copies, see Algorithm 3 p.97) slows down `MMMul5-BLAS`. This fact is more evident in a heavily parallel execution because the execution time of `dgemm` is one order of magnitude smaller. Then, as the matrix dimension grows, the ratio perceptibly tends to 5. Kernels `MMMul5frn` and `MMMul5bfrn` are still much less efficient than the BLAS implementation, but to a lesser extend than in the sequential case. The sudden decrease of the ratio around dimension 1,000 is related to the time increase of the reference `dgemm` already discussed.

To compare the best implementations, `MMMul5bfrn-avx` is faster than the non-guaranteed `MMMul5-BLAS` kernel for matrices up to dimension 2,500. After this value, `MMMul5bfrn-avx` gets slower and slower: this is due to the fact that the blocking strategy is limited to the first level of cache. An improved version with several levels of blocking and with thresholds tuned to the underlying cache hierarchy should remain competitive with the BLAS implementation for larger matrices.

5.4 Scalability

Finally, we analyze the *strong scalability* of the computation kernels, that is, their behavior when the number of threads rises while the matrix dimension remains constant. One important metric in this context is the measure of *efficiency*. By definition, the efficiency of a parallel implementation executed with p threads is the ratio $T_1/(p \times T_p)$, where T_1 is the sequential execution time, and T_p is the execution time using p execution threads. It equals one when the parallelism is perfect.

Figure 3 presents the measures of execution time (left) and scalability (right) of the `MMMul5-BLAS` and `MMMul5bfrn-avx` kernels, along with the corresponding values for `dgemm` implementation of the MKL. The input factors of the products are square matrices of dimension 1024, and the number of threads varies from 1 to 32.

The analysis of these measures explains why the `MMMul5bfrn-avx` implementation, which is slower than `MMMul5-BLAS` in sequential execution, becomes superior at high level of parallelism. In fact, the former implementation is about 1.4 times slower than the BLAS implementation with one thread, its execution time is comparable with 8 threads, and 1.5 times less with 32 threads. This phenomenon is more explicit in the left part of Figure 3: up to one thread per processor (4 threads), the efficiency of MKL `dgemm` is higher than 90%, then it falls to low levels, near 30%, with 32 threads. This is due to the fact that the memory bandwidth is shared among cores in a processor, and this resource rapidly becomes a limiting factor as the number of threads increases. The `MMMul5-BLAS` kernel, which is depends on the MKL `dgemm` for its more intensive

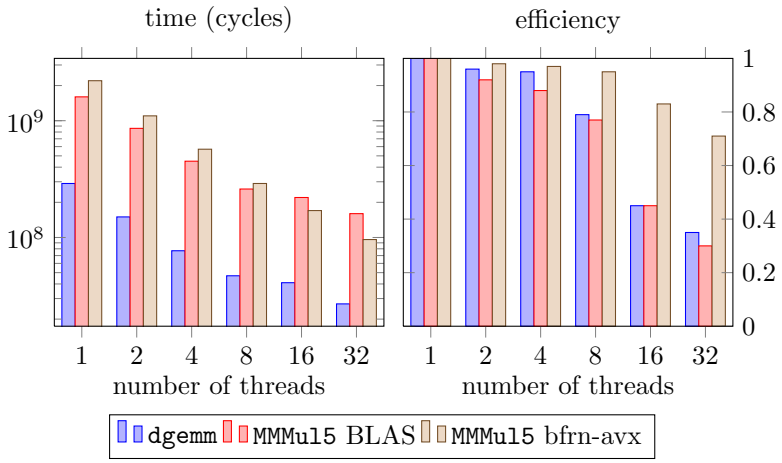

Figure 3: Scalability – 1024-by-1024 matrices.

computational part, exhibits the same significant decrease in efficiency. However, the behavior of `MMMul5bfrn-avx` demonstrates the possibility of improvement. In contrast to the BLAS implementation, the efficiency of `MMMul5bfrn-avx` decreases slowly and never drops below value of about 70% with 32 threads. The efficiencies of `MMMul5frn` and `MMMul5bfrn`, not shown here, follow the same scheme. This can be explained by the merging of the computations of $M_{\boldsymbol{A}} \cdot M_{\boldsymbol{B}}$, $\rho_{\boldsymbol{A}} \cdot \rho_{\boldsymbol{B}}$, $|M_{\boldsymbol{A}}| \cdot |M_{\boldsymbol{B}}|$, and $|\rho_{\boldsymbol{A}}| \cdot |\rho_{\boldsymbol{B}}|$ in the fused loop versions. It saves 3 successive calls to `dgemm`, which are as many global synchronization points in a parallel execution. On the contrary, the threads in the BLAS implementation have to wait for the latest ones among them, that are slowed down by concurrency to memory resource or system interruptions for instance, before entering the remaining computation.

6 Conclusion

The new results presented in Section 4 are algorithms for interval matrix multiplication that are actually 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 recurring question of the efficiency of these algorithms, since poor performance compared to the performance of (non-guaranteed but fast) floating-point algorithms reduces their value. Indeed, 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.

First, develop interval algorithms based on well-established numerical bricks (such

as the `gemm` of BLAS in this work), to benefit from their optimized implementation. A second step could be to convince developers and vendors of these bricks to clearly specify their behavior, especially with regards to rounding mode. However, we have observed that the above may not suffice, as this was the case with the issue of the order in which floating-point operations should be performed at each call.

The approach adopted in our paper has been 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 here to compute "simultaneously" $A \cdot B$ and $|A| \cdot |B|$. As can be observed from the experimental results, this approach pays off, as it makes it possible to guarantee the inclusion property — which is the fundamental property and strength of interval algorithms — and to get performance. To obtain even better performance, 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 has been done here. This would prevent the loss of performance when the matrix size increases too much. However, following such a methodology requires the programmer to have skills both in interval analysis and in HPC programming, which is rare.

The above methodology could lead to the development of a kind of Interval-BLAS, offering various algorithms with different accuracies (at least theoretically established) and good efficiency. The existence of such a library would be valuable to promote the wide use of interval algorithms, while alleviating the burden on the user of these algorithms to get performance.

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/blas/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. Baker Kearfott, Mitsuhiro T. Nakao, Arnold Neumaier, Siegfried M. Rump, Sergey P. Shary, and Pascal van Hentenryck. Standardized notation in interval analysis. *Computational Technologies*, 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.