Time-Domain BEM for the Wave Equation: Optimization and Hybrid Parallelization

Berenger Bramas[1], Olivier Coulaud[1], and Guillaume Sylvand[2]

[1] Inria Bordeaux, Sud-Ouest, 33405 Talence, France
[2] Airbus Group Innovations, Applied Mathematics and Simulation, Toulouse, France
{Berenger.Bramas,Olivier.Coulaud}@inria.fr, Guillaume.Sylvand@eads.net

Abstract. The problem of time-domain BEM for the wave equation in acoustics and electromagnetism can be expressed as a sparse linear system composed of multiple interaction/convolution matrices. It can be solved using sparse matrix-vector products which are inefficient to achieve high Flop-rate. In this paper we present a novel approach based on the re-ordering of the interaction matrices in slices. We end up with a custom multi-vectors/vector product operation and compute it using SIMD intrinsic functions. We take advantage of the new order of the computation to parallelize in shared and distributed memory. We demonstrate the performance of our system by studying the sequential Flop-rate and the parallel scalability, and provide results based on an industrial test-case with up to 32 nodes.

Keywords: Boundary element method (BEM), time domain, sparse matrix-vector product (SpMV), shared/distributed memory parallelization, SIMD.

1 Introduction

Airbus Group Innovations is an entity of Airbus Group devoted to research and development for the usage of Airbus Group divisions (Airbus Civil Aircraft, Airbus Defence & Space, Airbus Helicopters). The numerical analysis team has been working for more than 20 years on integral equations and boundary element methods for wave propagation simulations. The resulting software solutions are used on a daily basis in acoustics for installation effects computation, aeroacoustic simulations (in a coupled scheme with other tools), and in electromagnetism for antenna siting, electromagnetic compatibility or stealth. Since 2000, these frequency-domain Boundary Element Method (BEM) tools have been extended with a multipole algorithm (called Fast Multipole Method) that allows to solve very large problems, with tens of millions of unknowns, in reasonable time on parallel machines. More recently, H-matrix techniques have enabled the design of fast direct solvers, able to solve with a very high accuracy problems with millions of unknowns without the usual drawback associated with the iterative solvers (no control on the number of iterations, difficulty to find a good preconditioner, etc.). At the same time, we are working on the design and optimization of time domain BEM (TD-BEM) that allows to obtain with only one calculation the equivalent results of many frequency-domain computations. In this paper, we do not focus on the mathematical formulation of this TD-BEM (based on [2]), but rather on the parallel implementation of the algorithm.

In [3], the authors have implemented a TD-BEM application and their formulation is similar to the one we use. They show results up to 48 CPU and rely on sparse matrix-vector product without giving details on the performance. In [4], the author uses either multi-GPU or multi-CPU parallelization and accelerates the TD-BEM by splitting near field and far field. In [6], they give an overview of an accelerated TD-BEM using Fast Multipole Method. The paper does not contain any information on the sequential performance or even the parallelization which makes it difficult to compare to our work.

The optimization of the Sparse Matrix-Vector product (SpMV) operator has been widely studied because this is an essential operation in many scientific applications. Our work is not an optimization or an improvement for the general SpMV because we use a custom operator that matches our needs. Nevertheless, the optimizations of our implementation have been inspired by the historical work on SpMV which are the reordering of rows/columns, the management of the memory accesses, the blocking of the contiguous data or the data reuse, see [7],[8],[10],[9],[11],[12]. The performance is limited by the memory access pattern, the memory bandwidth and the instruction pipelining. It achieves 20% of the peak performance on common $X86$ architecture.

This paper addresses two major problems of the TD-BEM solver. First, we by-pass the low performance of SpMV by reordering the computation and by using a custom multi-vectors/vector product. Second, based on this new ordering we propose novel parallelization strategies for shared and distributed memory platforms.

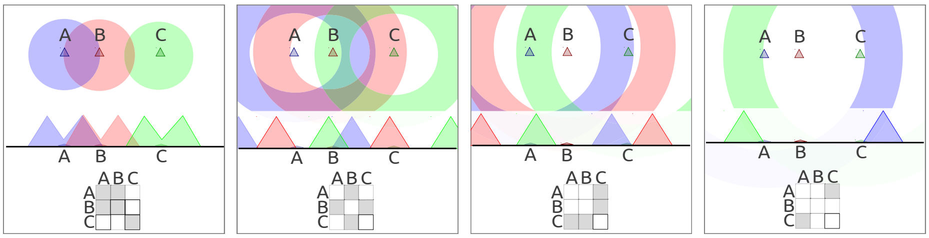
The rest of the paper is organized as follows. Section 2 provides background and mathematical formulation of the problem. Section 3 describes the new organization of computations and the multi-vectors SIMD operator. Section 4 details the parallelization strategies inherited from the new computational order. Finally, in Section 5 we provide an experimental performance evaluation of our multi-vectors/vector operator and of the different parallelization strategies.

2 Formulation

Our formulation has been originally defined in [2] but in order to keep this paper self-explanatory, we introduce the relevant aspects of the TD-BEM. An incident wave w with a velocity c and a wavelength λ is emitted on a boundary Ω. This surface Ω is discretized by N unknowns. The problem is also discretized in time with a step Δt and a finite number of iterations driven by the frequency study. In fact, increasing the number of time steps improves the results towards the bottom of the frequency range considered. At iteration time $t_n = n\Delta t$, the vector l^n contains the illumination of w over the unknowns from one or several emitters. The wave illuminates the location where the unknowns are defined and is reflected by these ones over the mesh. It takes a certain amount of time for the waves from the emitter or an unknown to illuminate some others. This relation is characterized by the interaction/convolution matrices M^k. A matrix M^k contains the interactions between unknowns that are separated by a distance around $k.c.\Delta t$ and contains zero for unknowns that are closer or further than this. These $N \times N$ matrices, where N is the number of unknowns, are positive definite and sparse in realistic configuration. They have the following properties:

- The number of non-zero values for a given matrix M^k depends on the structure of the mesh (the distance between the unknowns) and the physical properties of the system c, λ and Δt.
- For $k > K_{max} = 2 + \ell_{max}/(c\Delta t)$, with $\ell_{max} = max_{(x,y)\in\Omega\times\Omega}(|x-y|)$ the maximum distance between two unknowns, the matrices M^k are null.

The construction of these matrices is illustrated in Figure 1. The matrices are filled with values depending on the delay taken by a wave emitted by an unknown to pass over another one.



(a) M^0 (b) M^1 (c) M^2 (d) M^3

Fig. 1. Example of M^k matrices for three unknowns A, B, C in $1D$. A wave emitted from each unknown is represented at each time step. When a wave is around an unknown, a value is added in the matrix which is symbolized by a gray square. All matrices M^k with $k > 3$ are zero since the longest distance between elements is lower than $3.c.\Delta t$.

Convolution system. Using the convolution matrices M^k, and the incident wave l^n emitted by a source on the mesh, the objective is to compute the state of the unknowns a^n at time n for a given number of time iterations. The problem to solve at time step n is defined in Equation (1)

$$\sum_{k\geq 0}^{K^{max}} M^k \cdot a^{n-k} = l^n \, . \tag{1}$$

Equation (1) can be rewritten as in Equation (2) where the left hand side is the state to compute and the right-hand side is known from the previous time steps and the test case definition

$$a^n = (M^0)^{-1} \left(l^n - \sum_{k=1}^{K_{max}} M^k \cdot a^{n-k} \right) . \tag{2}$$

Solution algorithm. The solution is computed in two steps. In the first step, the past is taken into account using the previous values of a^p with $p < n$ and the interaction matrices as shown in Equation (3). The result s^n is subtracted from the illumination vector, see Equation (4)

$$s^n = \sum_{k=1}^{K_{max}} M^k \cdot a^{n-k} \,, \tag{3}$$

$$\widetilde{s}^n = l^n - s^n \,. \tag{4}$$

In the second step, the state of the system at time step n is obtained by solving the following linear system where $\widetilde{s}^n$ is the right-hand side

$$M^0 a^n = \widetilde{s}^n \,. \tag{5}$$

The first step is the most expensive part, from a computational standpoint. The solution of Equation (5) is extremely fast, since the matrix M^0 is symmetric, positive definite, sparse and almost diagonal. One can solve it using a sparse direct solver for example.

Context of the application. Our application is a layer of an industrial computational work-flow. We concentrate our work on the solution algorithm and we delegate to some black-boxes the generation of the interaction matrices and the direct solver. Moreover, in our simulations the meshes are static and all the interaction matrices and the pre-computation needed by the direct solver are performed once at the beginning. The most costly part of our algorithm is the computation of the right-hand side s^n. Our resulting implementation will replace a legacy version developed by Airbus Group Innovation which performs the solution algorithm using SpMV.

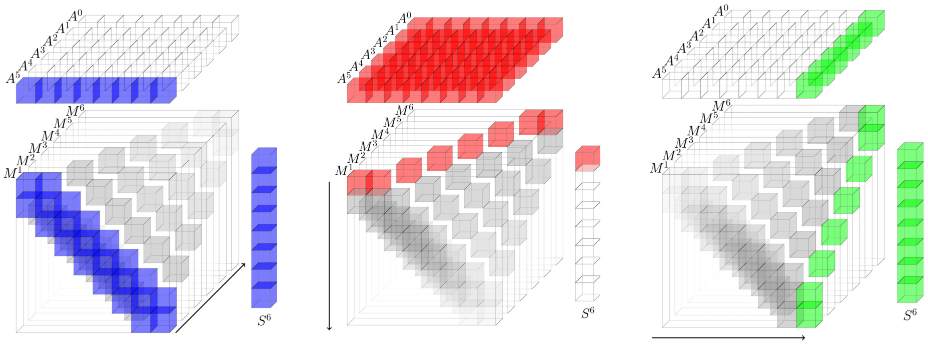
3 Summation Algorithm

3.1 Summation Ordering

We refer to the process of computing s^n as the summation stage. The summation uses the interaction matrices M^k and the past values of the unknowns a^{n-k}. A natural implementation of this computation is to perform K_{max} independent SpMV. That is implemented with four nested loops. The first loop is over the time step denoted by index n. The second loop is over the interaction matrices and is controlled by index k in our formulation and goes from 1 to K_{max}. Finally, the two remaining loops are over the rows and the columns of the matrices and are indexed by i and j respectively. The indices i and j cover the unknowns and go from 1 to N. The complete equation is written in Equation (6) where all indexes n, k, i and j are visible.

$$1 \leq i \leq N, s^n(i) = \sum_{k=1}^{k_{max}} \sum_{j=1}^{N} M^k(i,j) \times a^{n-k}(j) \tag{6}$$

In term of implementation, there is no need to keep the outer loop on index k and two other orders of summation are possible using i or j. The three possibilities are represented in Figure 2 where all interaction matrices M^k are shown one behind the other and represented as a $3D$ block. This figure illustrates the three different ways to access the interaction matrices according to the outer loop index. The natural approach using k is called by *front* and usually relies on SpMV. We propose to use a different approach called by *slice* using j as outer loop index. One can see the data access pattern of the interaction matrices in *slice* which is illustrated by Figure 2c.



A^0 A^1 A^2 A^3 A^4 A^5
M^1 M^2 M^3 M^4 M^5 M^6
S^6

(a) Front (k) (b) Top (i) (c) Slice (j)

Fig. 2. Three ways to reorder the computation of s^n with current time step $n = 6$, number of unknowns $N = 8$ and $K_{max} = 6$. For *front* the outer loop is on the different M^k matrices. For *top* the outer loop is over the row index of M^k and s^k. For *slice* the outer loop is over the column index of M^k.

3.2 Slice Structure

We use the word *slice* to name the data that are used when the outer loop index of the summation is j. A $Slice^j$ is composed of the concatenation of each column j of the interaction matrices $[M^1(*,j)\, M^2(*,j) \dots M^{K_{max}}(*,j)]$. Therefore, a slice is a sparse matrix of dimension $(N \times (K_{max} - 1))$. It has a non-zero value at line i and column k if $d(i,j) \approx k \cdot c \cdot \Delta t$, where $d(i,j)$ is the distance between the unknowns i and j. This definition is induced by the relation $M^k(i,j) = Slice^j(i,k)$. From the formulation, an interaction matrix represents the interaction between the unknowns for a given time/distance k. Whereas a $Slice^j$ represents the interaction that one unknown j has with others over the time. This provides an important property to the sparse structure of a slice: **the non-zero values are contiguous on each line**. In fact, it takes several iterations for a wave to cross over an unknown. In other words, for a given row i and column j all the interaction matrices M^k that have a non zero value at this position are consecutive in index k. In the slice format, it means that each slice has one vector per line but each of this vector may start at a different column k. If it takes p time steps for the wave from j to cross over i, then $Slice^j(i,k) = M^k(i,j) \neq 0$, $k_s \leq k \leq k_s + p$ where $k_s = d(i,j)/(c\Delta t)$.

3.3 Slice Computation

The Figure 3a shows a slice and how the values are contiguous on each line. It is natural to use level 1 BLAS dot-product instead of SpMV in order to take advantage of this particular structure. Therefore, for the entire summation defined in Equation (6), there are $N \times N$ dot-products to compute s^n (N dot-products per slice and N slices). However, the level 1 BLAS functions are memory bound and cannot achieve a high Flop-rate. In fact, for a vector of length v, we need to load $2.v+1$ floating point values to perform $2.v$ floating point operations (Flop). In order to increase the ratio of Flop against loaded data, we propose to compute several summation vectors together, see Figure 3b. By computing a group of n_g vectors s^*, we can use the matrix-vector product which is a level 2 BLAS operation. However, to compute the summation vector s^n at time n we need the past results from a^{n-1} to $a^{n-K^{max}}$. When we group, we also work on the future summation s^{n+1} which requires a^n to $a^{n-K^{max}+1}$. But a^n has not been computed yet since it needs s^n which is involved in the current process. That is why we need to replace the values of a^n by zero in the past values matrix. A similar strategy is requested for the other vectors of the group, and the vector s^{n+n_g-1} has its part of the past matrix starting with $n_g - 1$ zeros. Therefore, the past values matrix has a triangle of zero under the diagonal of the first rows. At time n, the algorithm computes $N \times N$ matrix-vector products and obtains n_g summation vectors where only the first one is complete. Like in the original algorithm, a direct solver gives a^n from $\widetilde{s}^n$ (Equations (4) and (5)). Because the

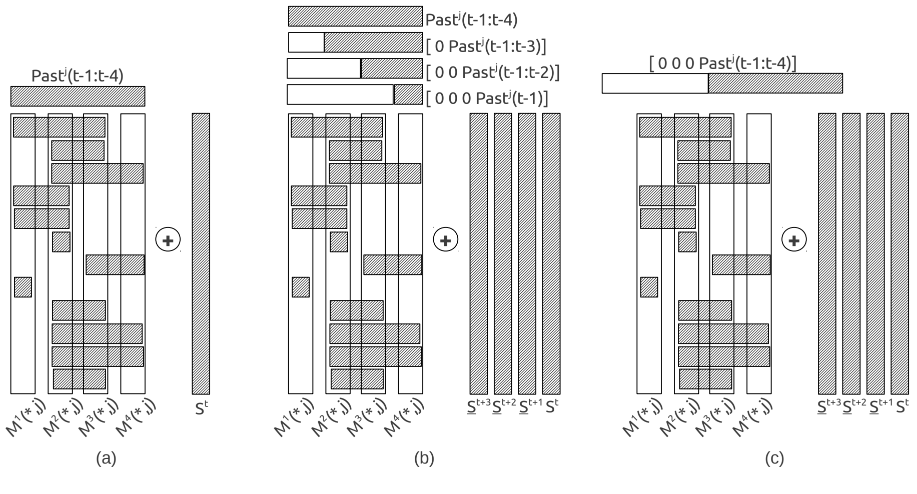


Fig. 3. Three ways of computing a slice product: (a) using dot-products, (b) by grouping with $n_g = 4$ and using matrix-vector product and (c) by grouping $n_g = 4$ and using custom multi-vectors/vector product

values of a^n were replaced by zeros in the summation the algorithm needs to update the incomplete summation vectors. The algorithm computes the action of the current values of a^n (at time step n) on the partial summation vectors from s^{n+1} to s^{n+n_g-1} (corresponding to future time steps $n+1$ to $n+n_g-1$) using SpMV and the n_g first interaction matrices. This operation is called radiation and it has to be repeated n_g times at each iteration.

The past matrix which is used in the slice computation using matrix-vector product has a particular structure. Each column is a copy of the previous column shifted by one and padded with zero (as illustrated on Figure 3b). Thus, instead of storing these data in a matrix of size $n_g \times K^{max}$, it is possible to use a special vector of size $n_g + K^{max} - 1$ with the values of a^{n-1} to $a^{n-K^{max}}$ at its end and with $n_g - 1$ zeros at its beginning as shown in Figure 3c. By grouping n_g vectors and using the special vector, we improve the ratio of Flop against loaded data: for a slice vector of length v, we need to load $2 \cdot v + 2 \cdot n_g - 1$ floating point values and we can perform $2 \cdot v \cdot n_g$ Flops. In such configuration we should be able to call an external matrix-vector product with a leading dimension of one for the past matrix. However, most of the BLAS libraries check the validity of the leading dimension and one is not a correct value. Moreover, a general matrix-vector product cannot take completely advantage of the pattern of the special past vector.

That is why we propose an implementation of an optimized operator to perform this operation and we refer to it as the multi-vectors/vector product. In our implementation, we reduce the memory access by re-using the past values, see Algorithm 1 that computes just one row in the set of output vectors s^n, s^{n+1}, ..., s^{n+n_g-1} .

Algorithm 1. Multi vectors/vector product

```
Data: $n_g$ the number of result vectors to compute simultaneously (should be $\geq 2$)
function MultiVectorsVector(vec[$SIZE\_VEC$], past[$SIZE\_VEC + n_g - 1$]) : res[$n_g$]
    register res[$n_g$] = 0;
    // We store the first past values (to load them once)
    register buffer[$n_g$-1];
    for idxBuffer = 0 → $n_g$-2 do
        buffer[idxBuffer] = load(past[idxBuffer]);
    end
    // For all values in the vec
    for idxVec = 0 → SIZE_VEC-1 do
        // Copy the current vec value
        register value = load(vec[idxVec]);
        for idxRes = 0 → $n_g$-3 do
            res[idxRes] += value * buffer[idxRes];
            // Shift the buffer value for the next idxVec loop
            buffer[idxRes] = buffer[idxRes+1];
        end
        res[$n_g$-2] += value * buffer[$n_g$-2];
        // Load a new value from the past vector
        buffer[$n_g$-2] = load(past[idxVec+$n_g$]);
        res[$n_g$-1] += value * buffer[$n_g$-2];
    end
    return res ;
```

4 Parallelization Strategies

4.1 Distributed Memory Parallelization

The parallelization over distributed memory is realized using Message Passing Interface (MPI) [16] and we name a MPI process a *process*. A slice interval is assigned to each process. This interval from j_{start} to j_{end} can be obtained in different ways: for example by dividing the number of N slices equally or by taking into account the amount of work in each slice. Each process needs to have the past values of the unknowns which match its slice interval. In a first stage, each process computes a part of the summation vectors without communicating with others. Then, all processes synchronize and call a sparse direct method to solve (5) and obtain the current solution a^n. With a number of threads per process equal to one, this algorithm is detailed in Algorithm 2.

At every iteration, the result is saved to disk for later work and it also has to be distributed to let each process have the current result for its interval j_{start} to j_{end}.

4.2 Shared Memory Parallelization

The straightforward parallelization in shared memory is implemented by splitting the slices computation and the radiation between threads. This is done using OpenMP *for pragma* [17] and it is detailed in Algorithm 2. If the number of threads per process is 1 and the parallelism relies on MPI only, we refer to the algorithm 2 as the Full-MPI implementation. If the number of threads is larger than 1, we refer to it as the Hybrid-MPI/OpenMP implementation.

5 Numerical and Performance Studies

5.1 Experimental Setup

Hardware configuration. We use up to 32 nodes and each node has the following configuration: 2 Quad-core Nehalem Intel Xeon X5550 at $2.66GHz$ and $24GB$ (DDR3) of shared memory.

Compiler and libraries. We use the Gcc 4.7.2 compiler and Open-MPI 1.6.5. The compilation flags are -m64 -march=native -O3 -funroll-loops -freorder-blocks-and-partition -ftree-vectorize -msse -msse2 -msse3 -mfpmath=sse. The direct solver is a state of the art solver Mumps 4.10.0 [15] which relies on Parmetis 3.2.0 and Scotch 5.1.12b. The calculation is performed in 64 bit arithmetic.

Test case. The test case is an airplane composed of 23 962 unknowns shown in Figure 4. The simulation should perform 10 823 time iterations. There are 341 interaction matrices. The total number of non-zero values in the interaction matrices, except M^0, is 5.5×10^9. For one iteration the total amount of Flops to compute the summation s^n is around $11\,GFlops$. If we consider that the direct

Algorithm 2. Complete simulation with Hybrid-MPI/OpenMP parallelization

```
Data: $Slices[N]$ the interaction matrices in slice/vectors shape. Each process is working on
      an interval [j_start; j_end] that cover the entire slices.
Result: $PastValues[j\_end - j\_start + 1][NB\_STEPS + n_g - 1]$ the state of the unknowns
        for all time step
begin
    // Direct Solver initialization (factorize/inverse $M^0$)
    $invM^0$_handle = direct_solver($M[0]$);
    // For all time step with progression by $n_g$
    for $n = 0 \to NB\_STEPS\text{-}1$ by $n_g$ do
        S[$n_g$][N] = 0;
        // Compute $n_g$ vectors with each slices in my interval
        #pragma omp parallel reduce(+:S);
        for j = j_start → j_end do
            foreach Vec v in Slices[j].blocks do
                S[:][v.row] += MultiVectorsVector(v.values, PastValues[j][v.col - $n_g$ + 1
                :v.col + v.length]) ;
            end
        end
        // Finalization
        for idx = 0 → $n_g$-1 do
            distributed_reduce(S[$n_g$ - idx -1][:]);
            $a^n$ = solve($invM^0$_handle, L[n+idx][:] - S[$n_g$ - idx - 1][:]);
            master saves $a^n$ to disk;
            // Copy result in Pastvalues format
            PastValues[j_start:j_end][NB_STEPS - n - 1] = $a^n$[j_start:j_end];
            // Radiation
            #pragma omp parallel;
            for past = idx + 1 → $n_g$-1 do
                S[$n_g$ - past][:] += SpMV($M^{past-idx}$[j_start:j_end],$a^n$[j_start:j_end];
            end
        end
    end
end
```

solver has the cost of a matrix-vector product, the total amount of Flop for the entire simulation is 130 651 *GFlop*. Storing all the data of the simulation takes more than 70 *GB*. Our application can execute out-of-core simulations, but we concentrate our study on in-core executions. We need at least 4 nodes to have the entire test case fitting in memory.

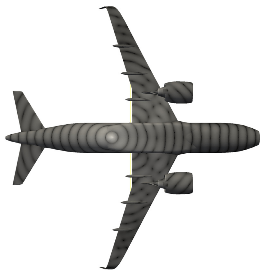


Fig. 4. Illustration of the Airplane test case

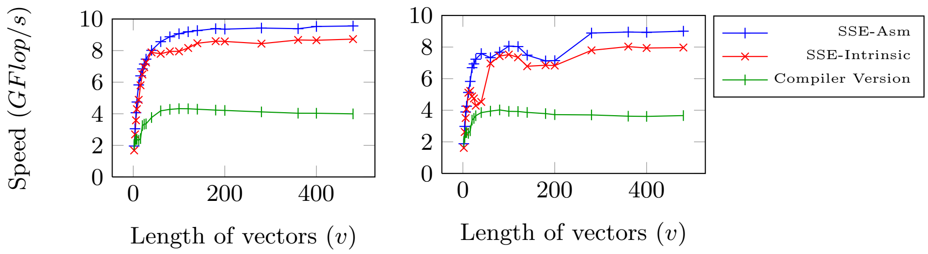
Parallel Efficiency. Usual parallel efficiency is defined by $e_n = T_1/(T_p * p)$ where T_1 is the sequential elpased time to execute the simulation and T_p the elapsed time using p cores. In our case, we use a modified version of the definition because we use at least 4 nodes (to remain in-core) and never execute the simulation sequentially. Using 1 core as a reference would artificially improve efficiency, since we would compare sequential out-of-core computations with parallel in-core computations. Hence, we replace the sequential time T_1 by T_r the time taken by the lowest number of cores which gives the new efficiency formula $\widetilde{e}_n = (r * T_r)/(T_p * p)$ where r is the number of cores for the time reference.

5.2 Multi-vectors/Vector Product

We compare three implementations of the multi-vectors/vector product. We choose to have $n_g = 8$ as it is enough to by-pass the memory bandwidth limitation without paying too much extra cost in the radiation stage. The first implementation comes out of the Equation (6) and is implemented in C. Some important compilation flags are used in order to enable loop unrolling and the use of SSE instructions by the compiler. This is referred to as the Compiler Version implementation. The second version is written in C and comes out of the Algorithm 1. It is written with intrinsic SSE functions proposed by the compiler and SSE data types ($\_\_m128d$). We refer to it as the SSE-Intrinsic implementation. We have analyzed the assembly code the compiler has generated and we have considered that it is not optimal for both implementations. Thus, we have developed a third implementation in asm64 assembly to maximize the data re-use. With $n_g = 8$ it is possible to use all 16 SSE registers in order to read each value only once from the main memory. We refer to it as the SSE-Asm implementation. Figure 5 shows the Flop-rate for all three operators for different lengths of vector v. The two SSE based implementations are close but the SSE-Asm can achieve a slightly higher Flop-rate for large vectors. Both implementations suffer from small cache effects for $N_r = 1\,000$ and $v = 100$ (Figure 5a) and for $N_r = 20\,000$, $v = 25$ and $v = 80$ (Figure 5b). However, the length of the vectors of the slices in real test cases depends on Δt the time step, and the size of the elements on the mesh. In the airplane test case, each vector has a length between 1 and 15 and the average length is 9.5. In this configuration, the SSE-Asm implementation achieves $3.9\,GFlop/s$ per core (Compiler Version achieves $1.7\,GFlop/s$) for a peak performance of $10.64\,GFlop/s$.

5.3 Scalability

We compare the Full-MPI and the Hybrid-MPI/OpenMP implementations to compute the airplane test case. We use 4 to 32 nodes and 8 cores per node. In Figure 6 we give the total execution time and the parallel efficiency. The efficiency is worthy for both implementations but in terms of execution time, the Full-MPI is better. Even if the number of processes involved in the global communications becomes larger because there are 8 MPI processes on each node, there is no advantage to reduce this number by having one process per node

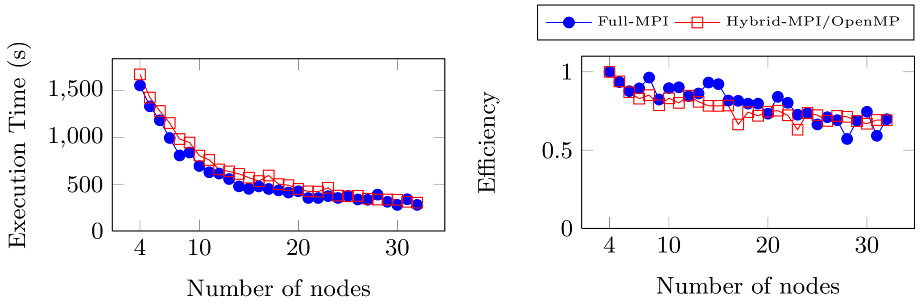


(a) Number of rows in slices N_r = 1 000

(b) Number of rows in slices $N_r = 20\,000$

Fig. 5. Performance evaluation in $GFlop/s$ for the multi-vectors/vector slice computation code for three implementation methods with $n_g = 8$. The test cases are slices of dimension $N_r \times v$.

and intra-node parallelism using threads. Figure 7 gives the percentages of time taken by the different operations. The time spent for the summation decreases as the number of nodes increases for both implementations. However, we can see that the Hybrid-MPI/OpenMP implementation exhibits more idle time than the Full-MPI when the number of nodes increases. In the Hybrid-MPI/OpenMP implementation some parts of the code are sequential, the threads share data, they parallelize small operations like the radiation for instance and the work is balanced statically between threads. In consequence, there are less MPI-processes in the Hybrid-MPI/OpenMP implementation but the threads are less balanced and they have to wait longer in the synchronization/reduction points.



(a) Execution time

(b) Parallel efficiency $\widetilde{e}_n$

Fig. 6. Execution time and parallel efficiency of the airplane simulation for the Full-MPI and the Hybrid-MPI/OpenMP implementations using 4 to 32 nodes, 8 CPU per node and $n_g = 8$

The previous application used by Airbus Group takes 13 500 seconds to compute the airplane simulation on 6 nodes. The new version presented in this paper takes only 1 200 seconds, which is around 10 times faster.

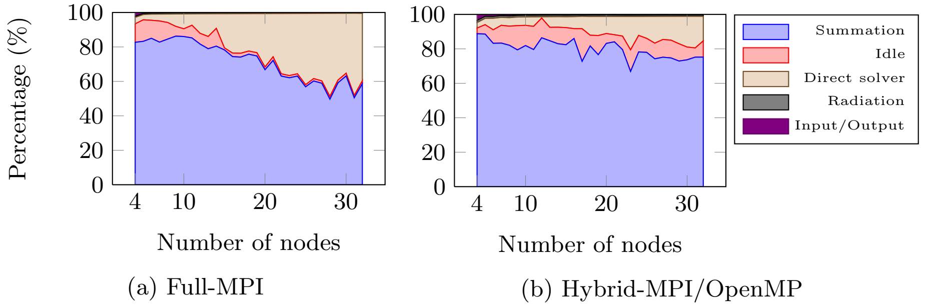


Fig. 7. Percentage of the time taken for the different operations to compute the airplane simulation for the Full-MPI and the Hybrid-MPI/OpenMP implementations using 4 to 32 nodes, 8 CPU per node and $n_g = 8$

6 Conclusion

We have presented a new parallelization and efficient implementation of a TD-BEM solver. We showed that the method scales efficiently and how the reordering of the computation leads to a good Flop-rate despite the sparse structure of the data. Moreover, our current application has a speedup of 10 against the previous implementation. In future work, we intend to compute larger simulations and in the longer term to use accelerators. We intend to investigate how a *Slice* product can be performed efficiently on accelerators using the abundant research that has been developed for the SpMV.

Acknowledgement. Experiments presented in this paper were carried out using the PLAFRIM experimental test bed. This work is supported by the Airbus Group Innovations, Inria and Conseil Régional d'Aquitaine initiative.

References

1. Liu, Y.J., Mukherjee, S., Nishimura, N., Schanz, M., Ye, W., Sutradhar, A., Pan, E., Dumont, N.A., Frangi, A., Saez, A.: Recent advances and emerging applications of the boundary element method. ASME Applied Mechanics Review 64(5), 138 (2011)
2. I. Terrasse, Résolution mathématique et numérique des équations de Maxwell instationnaires par une méthode de potentiels retardés, PhD dissertation, Ecole Polytechnique Palaiseau France (1993)
3. Abboud, T., Pallud, M., Teissedre, C.: SONATE: A Parallel Code for Acoustics Nonlinear oscillations and boundary-value problems for Hamiltonian systems, Technical report (1982), `http://imacs.xtec.polytechnique.fr/Reports/sonate-parallel.pdf`
4. Hu, F.Q.: An efficient solution of time domain boundary integral equations for acoustic scattering and its acceleration by Graphics Processing Units. In: 19th AIAA/CEAS Aeroacoustics Conference, ch. (2013), doi:10.2514/6.2013-2018

5. Langer, S., Schanz, M.: Time Domain Boundary Element Method. In: Marburg, S., Nolte (eds.) Computational Acoustics of Noise Propagation in Fluids - Finite and Boundary Element Methods, pp. 495–516. Springer, Heidelberg (2008)
6. Takahashi, T.: A Time-domain BIEM for Wave Equation accelerated by Fast Multipole Method using Interpolation, pp. 191–192 (2013), doi:10.1115/1.400549
7. Karakasis, V., Goumas, G., Koziris, N.: Perfomance Models for Blocked Sparse Matrix-Vector Multiplication Kernels. In: International Conference on Parallel Processing 2009, pp. 356–364 (2009), doi:10.1109/ICPP.2009.21
8. Nishtala, R., Vuduc, R.W.: When Cache Blocking of Sparse Matrix Vector Multiply Works and Why. In: Proceedings of the PARA 2004 Workshop on the State-of-the-art in Scientific Computing (2004)
9. Toledo, S.: Improving the memory-system performance of sparse-matrix vector multiplication. IBM Journal of Research and Development 41(6), 711–725 (1997)
10. Pinar, A., Heath, M.T.: Improving performance of sparse matrix-vector multiplication. In: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing. ACM (1999)
11. Yzelman, A.N., Bisseling, R.H.: Cache-Oblivious Sparse MatrixVector Multiplication by Using Sparse Matrix Partitioning Methods. SIAM Journal on Scientific Computing 31(4), 3128–3154 (2009), doi:10.1137/080733243
12. Vuduc, R.W., Moon, H.-J.: Fast sparse matrix-vector multiplication by exploiting variable block structure. In: Yang, L.T., Rana, O.F., Di Martino, B., Dongarra, J. (eds.) HPCC 2005. LNCS, vol. 3726, pp. 807–816. Springer, Heidelberg (2005)
13. Goto, K., Advanced, T.: High-Performance Implementation of the Level-3 BLAS, 117 (2006)
14. Morton, G.M.: A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. International Business Machines Company (1966)
15. Amestoy, P.R., Duff, I.S., L'Excellent, J.-Y.: MUMPS MUltifrontal Massively Parallel Solver Version 2.0 (1998)
16. Snir, M., Otto, S., et al.: The MPI core, 2nd edn (1998)
17. OpenMP specifications, Version 3.1 (2011), http://www.openmp.org