On the Block Wavelet Transform Applied to the Boundary Element Method

Henrique F. Bucher[1], Luiz C. Wrobel[2], Webe J. Mansur[3] and Carlos Magluta[3]

[1]Center For Computational Research, State University of New York at Buffalo,
9 Norton Hall, UB North Campus, Buffalo, NY, 14260, USA (henrique@bucher.com)

[2] Department of Mechanical Engineering, Brunel University,
Uxbridge UB8 3PH, Middlesex, UK (luiz.wrobel@brunel.ac.uk)

[3]Department of Civil Engineering, COPPE/Federal University of Rio de Janeiro,
PO Box 68506, CEP 21945-910, Rio de Janeiro, Brazil (webe@coc.ufrj.br;
magluta@labest.coc.ufrj.br)

Keywords: Wavelet transforms, boundary element method, data compression, fast solvers

Abstract. This paper follows an earlier work by Bucher *et al.* [1] on the application of wavelet transforms to the boundary element method, which shows how to reuse models stored in compressed form to solve new models with the same geometry but arbitrary load cases - the so-called virtual assembly technique. The extension presented in this paper involves a new computational procedure created to perform the required two-dimensional wavelet transforms by blocks, theoretically allowing the compression of matrices of arbitrary size. Details of the computer implementation that allows the use of this methodology for very large models or at high compression ratios are given. A numerical application shows a standard PC being used to solve a 131,072 DOF model, previously compressed, for 100 distinct load cases in less than 1 hour – or 33 seconds for each load case.

1. Introduction

The solution of very large problems arising from the Boundary Element Method (BEM) is still a big challenge for most industrial application developers due to the $\boldsymbol{O}(N^2)$ growth of the system matrices as the number of degrees of freedom in the model increases. Several methods were suggested [2][3][4][5] where this explosive growth is substituted by a less restrictive behavior of order $\boldsymbol{O}(NlogN)$. However, these methods require a complete rewriting of all routines used to perform the calculation of coefficients of the BEM matrices and, most often, also require the development of new companion solvers, upper bounds and respective spectral estimates.

However, reutilization of previous results obtained through extensive research during the last few decades is of importance for both academic researchers and software developers. For academic researchers it is important to have a solid theoretical basis so other researchers can verify new achievements independently. For developers, knowing extensively the key technologies used in the software is of prime importance so they can oversee any possible malfunction or even failure.

This work proposes a new methodology that extends achievements obtained in a previous work [1] to medium and large problems – of order 10^5 – while, at the same time, retaining all previous algorithms found in many textbooks, whose characteristics are well studied and documented. The key on this methodology is a block transform that acts as a black box, which can be seamlessly included in most classic source codes with just a small effort. Furthermore, the use of the Virtual Assembly (VA) technique empowers the block transform such that processed matrices can be reused in their compressed form to rapidly solve an arbitrary number of different load cases.

2. Review of the wavelet theory

Deriving a wavelet-like transform has been a goal for many mathematicians from as early as the beginning of the 20th century [6] but only in recent years this was summarized in a single, coherent theory [7][8]. The interested reader may refer to the cited works for an in-depth review of the theory but, for the sake of understanding how wavelet transforms can be used for numerical applications, it is sufficient to make a comparison with the Fourier transform.

First, both transforms are simple rotations in space and, therefore, they preserve dimensionality. The consequence is that they take a vector of N samples and produce another vector with exactly N coefficients. A positive feature of both transforms is that fast algorithms are easily available for forward and inverse operations. The Fast Wavelet Transform (FWT) is usually much faster than the Fast Fourier Transform (FFT) since the former does not require the computation of any transcendental function, just sums, subtractions and multiplications.

Therefore, both transforms are completely alike and share the same basic principles. Table 1 shows how this relationship goes even farther. The series expansion is basically the same but the wavelet transform is a double summation instead of a single summation as in the case of the Fourier transform. The reason is that, in the case of the Fourier transform, the variable j means frequency; the respective coefficient d_j can be interpreted as the amplitude of a sinusoidal function with infinite support. The wavelet coefficient $d_{j,k}$, however, can be viewed as the amplitude of a small wave-like perturbation, a wavelet; the usual example of wavelet is a sine wave multiplied by a Gaussian window. In this case, the variable j refers to the frequency of this wave and the variable k refers to its energy center in time.

The basis functions are, for Fourier, complex exponentials while for wavelets they can be any function that meet the wavelet conditions [8]. In spite of this, all basis functions are related to a single function called the “mother” wavelet, which generates the rest of them by simultaneous scaling and translation operations (see Table 1).

A group of conceptually-related mother functions is called a family; some examples of families are Daubechies, Coiflets, Symmlets and B-Splines [8]. Mother functions differ by their order, which is related to the number of coefficients of their respective FIR filter (a parameter of the FWT). Usually, the higher the order, the better the spectral quality but longer it takes to compute the FWT.

The evaluation of coefficients is basically the same - the integral and respective inverse operations, as previously mentioned, count with an $\boldsymbol{O(N.logN)}$ fast algorithm to speed-up their computation.

The orthogonality condition guarantees that reconstruction is perfect and energy is preserved after the transform.

Table 1 Comparison between the Fourier and wavelet series

Characteristic	Fourier Series	Wavelet series
Series expansion	$f(t)=\sum_{j=-\infty}^{\infty} d_j s_j(t)$	$f(t)=\sum_{j=-\infty}^{\infty}\sum_{k=-\infty}^{\infty} d_{j,k}\psi_{j,k}(t)$
Basis funcions	$s_j(t)=e^{i\frac{j\pi t}{L}}$	$\psi_{j,k}(t)$
"Mother" function	$s(t)=e^{i\frac{\pi t}{L}}$	$\psi(t)$
Relationship of the mother function with the rest of the basis functions	$s_j(t)=s(jt)$	$\psi_{j,k}(t)=2^{-j/2}\psi(2^{-j}t-k)$
Evaluation of coefficients	$d_j=\frac{1}{\sqrt{2L}}\int_{-L}^{L} f(t)s_j(-t)dt$	$d_{j,k}=\int_{-\infty}^{+\infty} f(t)\psi_{j,k}(t)dt$
Orthogonality condition	$\int_{-\infty}^{+\infty} s_j(t)s_m(t)dt=\delta_{jm}$	$\int_{-\infty}^{+\infty}\psi_{j,k}(t)\psi_{m,n}(t)dt=\delta_{jm}\delta_{pn}$
Energy conservation	$\int_{-\infty}^{+\infty}\lvert f(t)\rvert^2 dt=\sum_{j=-\infty}^{\infty}\lvert d_j\rvert^2$	$\int_{-\infty}^{+\infty}\lvert f(t)\rvert^2 dt=\sum_{j=-\infty}^{\infty}\sum_{k=-\infty}^{\infty}\lvert d_{j,k}\rvert^2$

3. Review of the virtual assembly technique

The BEM, in its direct formulation for potential problems, generates the following system of equations when the standard point collocation technique is used [9]:

$$\boldsymbol{Hu}=\boldsymbol{Gq} \tag{1}$$

The coefficients of the matrices ***G*** and ***H*** in the above equation result from the integration of the fundamental solution and its normal derivative within each element. Vectors ***u*** and ***q*** contain nodal values of the variable under consideration and its derivative in the normal direction.

According to the virtual assembly technique presented by Bucher *et al.* [1], the original system of equations (1) may be expressed as a function of the compressed boundary element matrices

$$\tilde{\boldsymbol{H}}=\boldsymbol{treshold}(\boldsymbol{WHW}^{\boldsymbol{T}}) \tag{2}$$

and

$$\tilde{\boldsymbol{G}}=\boldsymbol{threshold}(\boldsymbol{WGW}^{\boldsymbol{T}}) \tag{3}$$

such that an alternative form of the original system of equations (1) is generated:

$$\left(\left(\boldsymbol{W}^T\tilde{\boldsymbol{H}}\boldsymbol{W}\right)\boldsymbol{X}_{\boldsymbol{H}}-\left(\boldsymbol{W}^T\tilde{\boldsymbol{G}}\boldsymbol{W}\right)\boldsymbol{X}_{\boldsymbol{G}}\right)\boldsymbol{x}=\boldsymbol{f} \tag{4}$$

where ***W*** is the linear operator representing the orthogonal transform used to compress the original matrices ***H*** and ***G***, very often the wavelet transform. $\boldsymbol{X}_{\boldsymbol{G}}$ and $\boldsymbol{X}_{\boldsymbol{H}}$ are diagonal operator matrices such that their coefficients are

$$\left(\boldsymbol{X}_{\boldsymbol{H}}\right)_{i,j}=\begin{cases}\lambda_i^H & \text{if } i=j\\ 0 & \text{elsewhere}\end{cases}\qquad \left(\boldsymbol{X}_{\boldsymbol{G}}\right)_{i,j}=\begin{cases}\lambda_i^G & \text{if } i=j\\ 0 & \text{elsewhere}\end{cases} \tag{5}$$

λ^G and λ^H are operators responsible for applying boundary conditions and consequently generating the traditional system of equations $\boldsymbol{Ax}=\boldsymbol{b}$. Instead of exchanging rows and columns in the traditional procedure, we write each nodal value u and its derivative q as a function of the respective unknown x in the solution vector as

$$\begin{aligned}u&=\lambda^H x+u_0\\ q&=\lambda^G x+q_0\end{aligned} \tag{6}$$

This operation will allow solving the system using the two matrices, ***G*** and ***H***, in their compressed form without generating the system matrix ***A***, therefore justifying the expression “virtual assembling”. This technique produces a greater freedom for application of diverse boundary conditions, as shown in Table 2.

Table 2 Boundary conditions applied through the virtual assembly technique

Condition type	Expression	Implementation
Prescribed potential	$u=\overline{u}$	$\lambda^H x=0,\ \ u_0=\overline{u}$ $\lambda^G x=x,\ \ q_0=0$
Prescribed flux	$q=\overline{q}$	$\lambda^H x=x,\ \ u_0=0$ $\lambda^G x=0,\ \ q_0=\overline{q}$
Linear relationship	$q=a+bu$	$\lambda^H x=x,\ \ u_0=0$ $\lambda^G x=bx,\ \ q_0=a$
Generic non-linear	$q=f(u)$	$\lambda^H x=x,\ \ u_0=0$ $\lambda^G x=f(x),\ \ q_0=0$

Therefore, the new system matrix is recognized as

$$\boldsymbol{A}=\left(\boldsymbol{W}^T\tilde{\boldsymbol{H}}\boldsymbol{W}\right)\boldsymbol{X}_{\boldsymbol{H}}-\left(\boldsymbol{W}^T\tilde{\boldsymbol{G}}\boldsymbol{W}\right)\boldsymbol{X}_{\boldsymbol{G}} \tag{7}$$

and the force vector is calculated by

$$\boldsymbol{f}=-\left(\boldsymbol{W}^T\tilde{\boldsymbol{H}}\boldsymbol{W}\right)\boldsymbol{u_0}+\left(\boldsymbol{W}^T\tilde{\boldsymbol{G}}\boldsymbol{W}\right)\boldsymbol{q_0} \tag{8}$$

As this system of equations remains in the original space, all algebraic properties such as convergence rates and condition numbers are preserved except for the often-negligible distortion introduced by the thresholding operation.

Several papers have studied the final effect of thresholding on the residual. Koro and Abe [10][11] showed that there is an optimal threshold for Beylkin-type matrix compression such that the residual error is equal to the intrinsic error introduced by discretization of the system boundary.

In the case of the virtual assembly technique, it is clear that final errors on the residuals will be quite different depending on the boundary condition applied. This is easy to see in equation (7), where the system matrix is written as a combination of the original matrices ***H*** and ***G*** and the diagonal, boundary-dependent matrices ***X***. Thus, depending on the boundary condition applied, the dominant matrix will be either ***H*** or ***G***, and this should have an impact on the optimal threshold value.

Therefore, in the case of the virtual assembly technique, there is no optimal threshold to be used at the compression phase. However, if "optimal" means an average value for a high number of load cases, the thresholds suggested in [10] and [11] are in fact the best available estimates.

4. The block transform

Obtaining the compressed matrices (2) and (3) requires full integration of matrices ***H*** and ***G*** before the transform and threshold operations can be applied. As the size of the matrices increase quadraticaly with the number of degrees of freedom, memory consumption becomes a major concern. Gonzales *et al.* [12] presented a solution to this problem where a parallel version of the wavelet transform is introduced, which avoids the single computer memory limitation but does not decouple the relationship between problem size and total memory available at one instant in time.

An effective solution to deal with problems of arbitrary size, independently of the amount of memory available in the computer, is possible by using a novel algorithm developed in this paper, the Block Wavelet Transform (BWT), which allows the compression and reutilization of the compressed blocks in computers equipped with virtually any amount of memory. By partitioning the BEM matrices by blocks, memory requirements are exchanged by time requirements which are very often less restrictive. Blocks can then be freely resized to fit the maximum memory of each computer, therefore maximizing the use of available resources.

Despite sharing some very basic ideas with other wavelet algorithms with the same name used in image compression by Cetin *et al.* [13] and Huh *et al.* [14], the BWT is based on a completely new theoretical development. In those papers, the wavelet transform is used to compress blocks of images that will be saved and decompressed when needed. Decompression, however, is unacceptable since the size of the original matrix might be orders of magnitude larger than the available memory. Thus, the block compression methodology must include a strategy to use the blocks for solving the system in its compressed form, which is done by using a mapping matrix known as *adjacency matrix* in the finite element terminology [15][16]. The adjacency matrix allows the description of the original system matrix in a series where the coefficients are blocks and the basis functions are the adjacency matrices.

However, not all possible matrix partitions are valid. In other words, there must be a condition that guarantees that no block overlapping or lack of coverage will occur. This work advances beyond the basic adjacency matrix concept by defining a way of restraining the grid partition through a condition called the admissibility condition:

Partition admissibility condition. A block partition is valid if and only if *any* matrix $\boldsymbol{A}$ can be reconstructed from its respective blocks $\boldsymbol{A}_k$, k=1 to *M,* through the series

$$\boldsymbol{A} = \sum_{\boldsymbol{k}=1}^{\boldsymbol{M}} \boldsymbol{L}_{\boldsymbol{N},\boldsymbol{n_k}}^{\boldsymbol{row_k}} \boldsymbol{A_k} \left(\boldsymbol{L}_{\boldsymbol{N},\boldsymbol{n_k}}^{\boldsymbol{col_k}} \right) \tag{9}$$

The special adjacency matrices $\boldsymbol{L}$ that appear in this series are called *Insertion-Extraction* (IE) matrices to emphasize that these are linked to new matrices with special partitioning properties.

Each IE matrix is denoted by the symbol $\boldsymbol{L}_{N,n}^{i}$, where *i* is the index in the expansion series, *N* is the size of the original, full matrix (supposed to be square), and *n* is the size of the respective block. All elements in an IE matrix are zero but those in the diagonal that starts at column *i*, which are ones. This shape provides two important block properties, also illustrated in Figure 1:

Extraction property. A square block $\boldsymbol{A}_k$ of size *n*, located at the position (*row_k*,*col_k*) of a generic matrix $\boldsymbol{A}$, can be extracted by pre-multiplying the transposed version of the IE matrix $\boldsymbol{L}_{N,n}^{row_k}$ followed by post-multiplying the result by the IE matrix $\boldsymbol{L}_{N,n}^{col_k}$ such that

$$\boldsymbol{A}_k = \left(\boldsymbol{L}_{N,n}^{row_k} \right)^T \boldsymbol{A} \boldsymbol{L}_{N,n}^{col_k} \tag{10}$$

Insertion property. The "*k*" block extracted using the above extraction property can be restored to its original position (row_k,col_k) in the original, full matrix $\boldsymbol{A}$ by pre-multiplying it by the IE matrix $\boldsymbol{L}_{\boldsymbol{N},\boldsymbol{n}}^{\boldsymbol{row_k}}$ followed by post-multiplying the resulting matrix by the transposed version of the IE matrix LE $\boldsymbol{L}_{N,n}^{col_k}$:

$$\boldsymbol{A}_k^{NxN} = \boldsymbol{L}_{N,n}^{row_k} \boldsymbol{A}_k \left(\boldsymbol{L}_{N,n}^{col_k} \right)^T \tag{11}$$

Both properties can be better understood with the help of Figure 1.

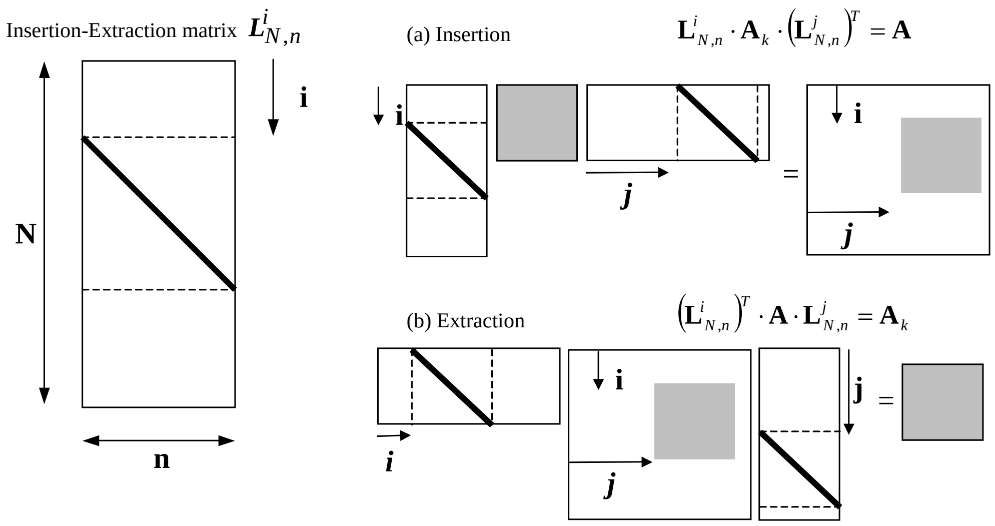

Insertion-Extraction matrix $L^{i}_{N,n}$
N
n
i
(a) Insertion
$\mathbf{L}^{i}_{N,n}\cdot\mathbf{A}_k\cdot\left(\mathbf{L}^{j}_{N,n}\right)^T=\mathbf{A}$
i
j
=
(b) Extraction
$\left(\mathbf{L}^{i}_{N,n}\right)^T\cdot\mathbf{A}\cdot\mathbf{L}^{j}_{N,n}=\mathbf{A}_k$
i
j
=

Figure 1 Properties of the Insertion-Extraction (IE) matrix

In this way, using the extraction property (10) for decomposition and the insertion property (11) for reconstruction, the square matrix ***A*** can now be sliced into *M* distinct blocks $\boldsymbol{A}_k$, *k* = 1 to *M*, each of them located in one particular position (row_k,col_k) and with size n_k.

4.1. Matrix-vector multiplication in blocks

The product ***b* = *A x*** between the original matrix ***A*** and a vector ***x*** can be performed in blocks by using the admissibility condition (9), when the following expression is obtained

$$\boldsymbol{b}=\boldsymbol{A}\;\boldsymbol{x}=\sum_{k=1}^{M}\boldsymbol{L}^{row_k}_{N,n}\quad\boldsymbol{A}_k\quad\left(\boldsymbol{L}^{col_k}_{N,n}\right)\;\boldsymbol{x} \tag{12}$$

The computation of such operation is in fact a fast procedure, which is better understood with the help of Figure 2.

$$\boldsymbol{A}\,\boldsymbol{x}=\sum_{k=1}^{M}\underbrace{\boldsymbol{L}^{row_k}_{N,n_k}\underbrace{\boldsymbol{A}_k\underbrace{\left(\boldsymbol{L}^{col_k}_{N,n_k}\right)^T\boldsymbol{x}}_{\textcircled{a}}}_{\textcircled{b}}}_{\textcircled{c}}$$

Loop in blocks (*k*)

(a) Extract the strip of length n_k beginning at position col_k of vector ***x***

(b) Multiply the result by block *k*

(c) Accumulate the result in an initially empty vector beginning at position row_k

Figure 2 Algorithm to calculate the matrix-vector multiplications in blocks

Global compression is obtained by compressing each block separately using the two-dimensional transform chosen, an operation that is represented by the following expression:

$$\tilde{\boldsymbol{A}}_k=\boldsymbol{W}_{n_k}\boldsymbol{A}_k\boldsymbol{W}^T_{n_k} \tag{13}$$

where $\tilde{\boldsymbol{A}}_k$ represents the transformed block already sparsified by thresholding it, *i.e.* by neglecting its most non-significant elements. As the transform is orthogonal, the original, non-transformed block can be recalculated by

$$A_k = W_{n_k}^T \tilde{A}_k W_{n_k} \tag{14}$$

The admissibility condition (9) for the adopted partition must hold for *any* square matrix $\boldsymbol{A}$ of size N. Therefore, applying equation (14) into the admissibility condition (9), the expression of the reconstructed, original matrix is obtained:

$$A = \sum_{k=1}^{M} L_{N,n_k}^{row_k} W_{n_k}^T \tilde{A}_k W_{n_k} \left(L_{N,n_k}^{col_k} \right)^T \tag{15}$$

This last equation expresses the original matrix $\boldsymbol{A}$ as a function of the compressed blocks $\boldsymbol{A_k}$. The above matrix-matrix multiplication will never be computed directly but is used to calculate the matrix-vector multiplications as required by the iterative solver. The new algorithm is better understood with the help of Figure 3.

$$Ax = \underbrace{\sum_{k=1}^{M} L_{N,n_k}^{row_k} \underbrace{W_{n_k}^T \underbrace{\tilde{A}_k \underbrace{W_{n_k} \underbrace{\left(L_{N,n_k}^{col_k} \right)^T x}_{\text{a}}}_{\text{b}}}_{\text{c}}}_{\text{d}}}_{\text{e}}$$

Loop into blocks (k)

(a) Extract the strip of length n_k starting at position col_k of vector $\boldsymbol{x}$

(b) Execute the direct transform on the resulting strip

(c) Multiply the result by the compressed block (sparse)

(d) Apply the inverse transform on the resulting vector strip

(e) Accumulate the result in an initially empty vector starting at position row_k

Figure 3 Algorithm to calculate the matrix-vector multiplication using the compressed blocks

This algorithm allows the matrix-vector multiplications required by the iterative solver using the blocks that were compressed by using an arbitrary partitioning grid. But when the adopted grid is chosen in such a way that all blocks have a constant size, then huge savings can be obtained. Given that all blocks in a regular grid have the same size n, the position of each block can be expressed by

$$\begin{aligned} row_i &= (i-1)n+1 \\ col_j &= (j-1)n+1 \end{aligned} \tag{16}$$

where the variables i and j span the range from 1 to $\sqrt{M}$, the square root of the total number of blocks, equal to the number of blocks per side. Using this regularity, expression (15) can be simplified to

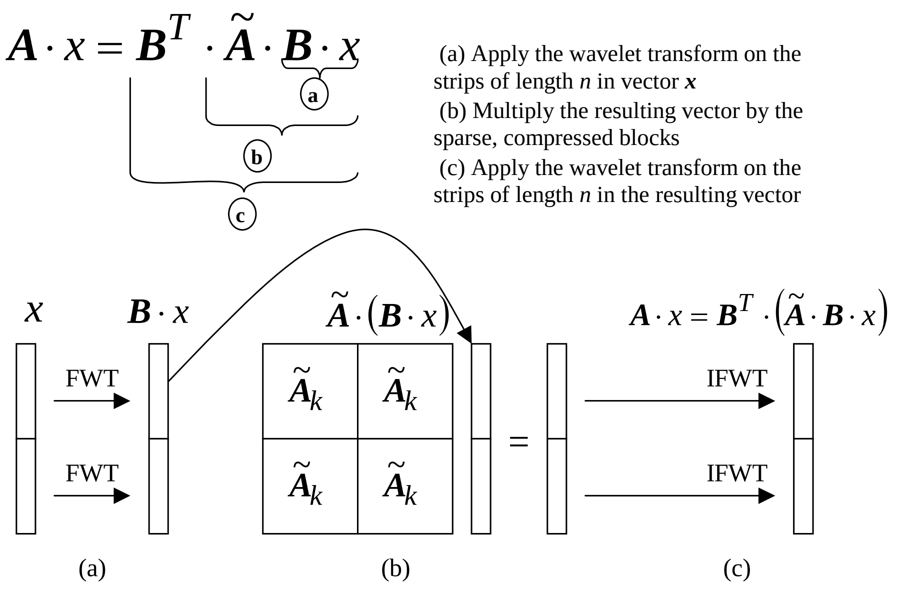
$$A = B^T \tilde{A} B \tag{17}$$

where the new transformation matrix $\boldsymbol{B}$ and the new compressed matrix $\tilde{\boldsymbol{A}}$ are given by

$$B = \sum_{i=1}^{M} L_{N,n}^{row_k} W_n \left(L_{N,n}^{col_k} \right)^T \tag{18}$$

$$\tilde{A} = \sum_{i=1}^{M} L_{N,n}^{row_k} \tilde{A}_k \left(L_{N,n}^{col_k} \right)^T \tag{19}$$

The latter results were obtained by noticing that the admissibility condition guarantees that the blocks do not overlap nor leave any matrix region uncovered. The new algorithm for calculation of the matrix-vector multiplications can then be easily computed as shown in Figure 4.



$A \cdot x = B^T \cdot \tilde{A} \cdot B \cdot x$
a
b
c
(a) Apply the wavelet transform on the strips of length *n* in vector ***x***
(b) Multiply the resulting vector by the sparse, compressed blocks
(c) Apply the wavelet transform on the strips of length *n* in the resulting vector
x
$B \cdot x$
$\tilde{A} \cdot (B \cdot x)$
$A \cdot x = B^T \cdot (\tilde{A} \cdot B \cdot x)$
FWT
FWT
$\tilde{A}_k$
IFWT
IFWT
(a)
(b)
(c)

Figure 4 Algorithm to perform the matrix-vector multiplication when the blocks are divided and compressed using a regular grid

The regular form of the matrix-vector multiplication algorithm is naturally faster than the generic form because it avoids one summation level.

5. Computer implementation

Two aspects of their respective computer implementation heavily affect the performance of the proposed algorithms. First, compressed blocks must be stored as a sparse matrix for which there are many formats available. Second, as the virtual assembly technique enforces the reusability of the compressed matrices to calculate several load cases, the compressed blocks must be saved in a persistent media when there is a choice of format to be used: text or binary.

There are two popular storage formats for sparse matrices available [17][18]: compressed row (CR) and compressed coordinate (CC).

The compressed coordinate format is very simple, as illustrated in Figure 5. The total size used by this format to store the sparse matrix is 16*ne* when 8-byte doubles and 4-byte integers are used, where *ne* is the number of non-zero coefficients in the original matrix. This format is suitable for very sparse matrices.

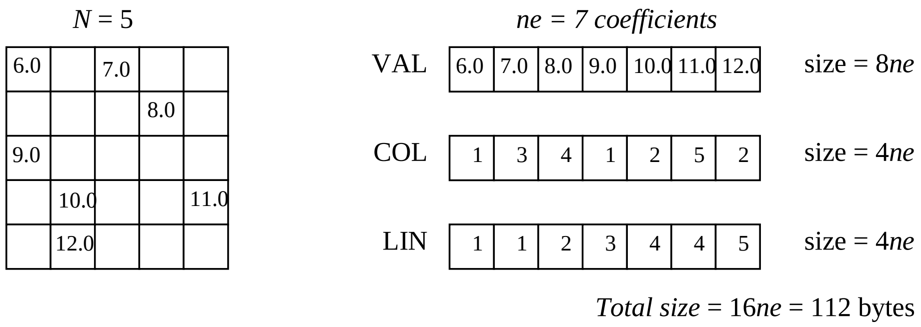


Figure 5 Sparse storage using the coordinate format (example)

The compressed-row format adds an additional level of compression to average and very populated sparse matrices. This format effectively reduces the storage space required by matrices whose number of coefficients *ne* is greater than the number of rows *N*, which is the case of most finite element models. The storage space required by the compressed-row format is $4N+12ne$ bytes (assuming 8-byte doubles and 4-byte integers).

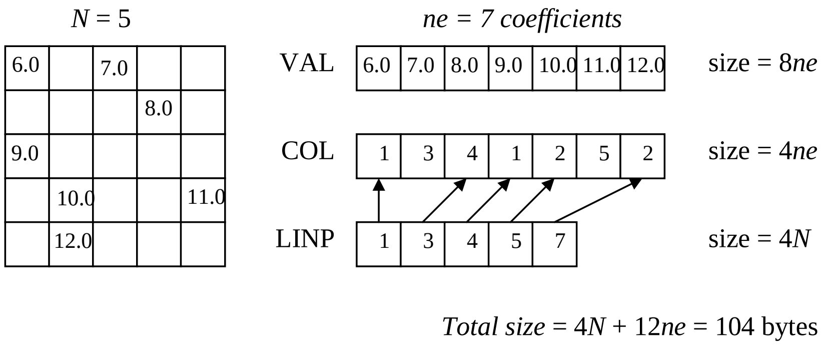


Figure 6 Sparse storage using the compressed-row format (example)

When the wavelet compression is applied, however, threshold values are set such that an optimum balance between compression ratio and error introduced in the solution is obtained. Therefore, the resulting sparse matrices may have more or less significant coefficients *ne* than their respective block size *N*, leaving open the choice of the best sparse format to adopt. Figure 7 shows the apparent compression ratio, *i.e.* the relation $8N^2/TotalSize$, where *TotalSize* is the amount of memory required to store the sparse structure, as a function of the number of significant coefficients *ne*. Several block sizes are plotted in Figure 7, ranging from 128 to 8192.

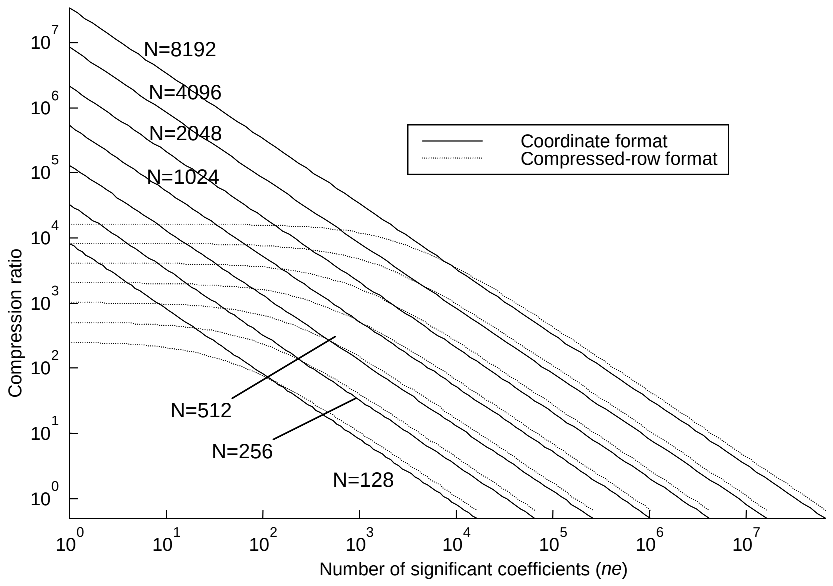


Figure 7 Compression ratio as a function of the number of coefficients in the sparse matrix for several block sizes and sparse formats

As shown by Figure 7, the compressed-row (CR) format is superior to the compressed coordinate (CC) format when the number of coefficients in the sparse matrix is more than its size, *i.e.* when the compression ratio is low. Therefore, the compressed-row format is not indicated to hold the sparse structures when the compression ratios are expected to be high. In this case, the coordinate format is better suited. Still, even in the case where there are many coefficients in the sparse matrix, the compression ratios resulting from the use of the coordinate format are not much lower than those obtained by the compressed-row format.

Despite the fact that double floating point values are required to obtain accurate results in most numerical applications, in many cases it is acceptable to introduce a certain amount of error by changing the type of the matrix elements from 64 bit doubles to 32 bit floats. This was found to introduce a relative RMS error less than 10^{-6} in the solution of all models studied in this paper.

An additional compression effect can be obtained by changing integer values from 32 bits to 16 bits. This will limit the compressed-row (CR) format to 65,536 coefficients maximum but will only limit the compressed coordinate (CC) to the maximum block size of 65,536, which is much less restrictive since most systems will already be limited to blocks of 8192 elements due to memory availability – 512 MB in this case. Figure 8 shows the effect of these changes in the resulting compression ratios.

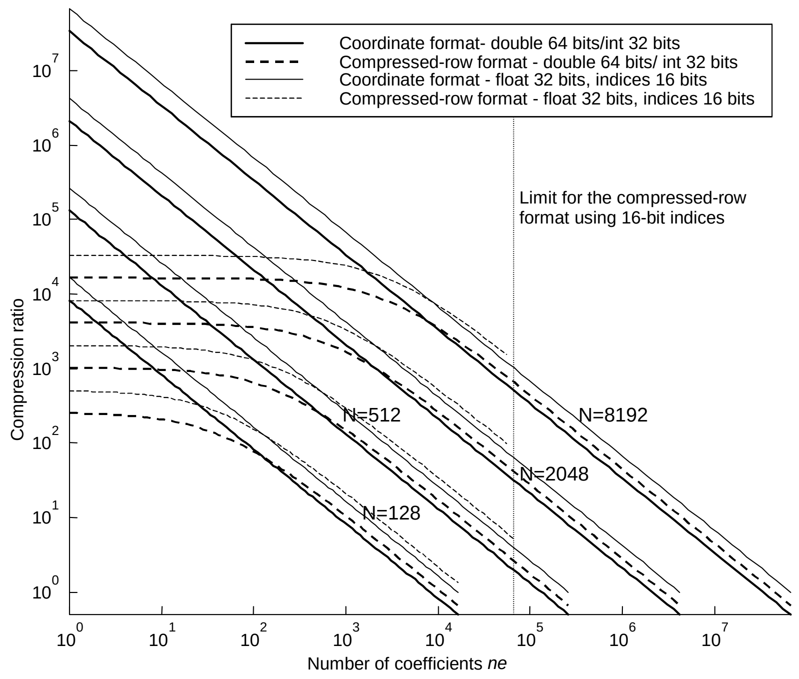


Figure 8 Additional compression effect caused by substitution of 64-bit doubles and 32-bit indices by 32-bit floats and 16-bit integers

Under very high compression ratios, very often no coefficients in a block are above the adopted threshold value, which results in an empty sparse structure. By removing the contribution of the block’s average plane, the matrix-vector multiplication will then always be represented, even if all the block coefficients after the transform operation lie below the specified threshold value.

The average plane π of a square matrix $\boldsymbol{A}$ is the one that minimizes the least squares condition

$$r = \sum_{i=1}^{N}\sum_{j=1}^{N}\left(A_{ij} - \pi(i, j)\right)^2 \tag{20}$$

where the values of the plane π at a point *(i,j)* are given by

$$\pi(i, j) = \pi_a \cdot i + \pi_b \cdot j + \pi_c \tag{21}$$

The coefficients π_a, π_b and π_c can be retrieved by solving the following 3×3 system of equations

$$\frac{N^2(N-1)}{6}\begin{bmatrix} 3(N-1) & 2(N-1) & 3 \\ 2(N-1) & 3(N-1) & 3 \\ 3 & 3 & N^2 \end{bmatrix} \cdot \begin{bmatrix} \pi_a \\ \pi_b \\ \pi_c \end{bmatrix} = \begin{bmatrix} S_X \\ S_Y \\ S_0 \end{bmatrix} \tag{22}$$

where S_X, S_Y and S_0 are matrix moments, calculated by

$$S_X = \sum_{i=1}^{N}\sum_{j=1}^{N}(i-1)A_{ij} \tag{23}$$

$$S_Y = \sum_{i=1}^{N}\sum_{j=1}^{N}(j-1)A_{ij} \tag{24}$$

$$S_0 = \sum_{i=1}^{N}\sum_{j=1}^{N} A_{ij} \tag{25}$$

Therefore, compression is performed not on the original block but on the block resulting from the subtraction of the average plane from the original block. The calculation of the average plane and its subsequent subtraction are often done very fast and do not contribute to significantly increase the amount of compression time.

The additional task of deciding the storage format to save the compressed blocks must still be studied with care since the original uncompressed matrices can easily reach the size of 500 Gigabytes. A binary format allows rapid I/O but it is not unique to all platforms. A text format tends to be more universal but it requires a lot more disk space, typically 5 to 10 times the binary format, and allows very slow read operations, typically 100 times slower.

The solution found in this work is the use of a universal binary format called XDR [19]. This format is heavily used in message passing APIs such as PVM. Just part of the specification was implemented, only enough to handle integer (16 and 32 bits) and floating point (double and float) types. Portability was tested on the following platforms: Intel PC/Windows (GNU GCC v2.95.2 and MS Visual Studio C++ 6 compilers), Sun/Solaris (GNU GCC v2.95.2), DEC Alpha/OSF (GNU v2.95.2 and Alpha C++ compilers). A series of I/O tests using a standard Intel PC computer were performed to test the difference between a text format and a binary format; their results are shown in Table 1.

Table 1 Times (in seconds) obtained for input and output of a vector from memory to the hard drive using the XDR format and a simple text format

Vector size	Binary (XDR)				Text			
	Floating point		Integer		Floating point		Integer	
	Write	Read	Write	Read	Write	Read	Write	Read
500 kb	0.08	< 0.01	0.07	< 0.01	0.82	0.59	0.64	0.20
5 Mb	0.66	0.11	0.88	0.22	10.27	6.98	8.68	2.58
50 Mb	9.82	0.83	10.66	0.82	95.58	73.93	81.68	31.42
1Gb	192.4	15.97	208.7	15.4	1902.3	1450.9	1631.5	620.2

The binary format presented an average performance 10 times superior to the text format for output. The largest difference occurs in the reading operation when the binary format is 100 times faster than the text, with the exception of reading times for integer numbers, when it is only 40 times faster.

A 1Gb sparse matrix in coordinate sparse format is composed of about 512Mb of floating point numbers and 512Mb of integer numbers, which will require 31.4 seconds to be read in binary format. If text format is used, this time raises to 34 minutes, a 65 times increase. The final implementation decision was then to adopt 16-bit, compressed coordinate format and store the resulting compressed blocks in the partially compliant XDR format described above.

The final computer implementation workflow is shown in figure 9, where a regular grid strategy was used for matrix partitioning.

Assembly Phase

```
1. Adopt a matrix partition
2. For each cell in the partition do
        2.1. Assemble blocks $G_k$ and $H_k$
        2.2. For each block $G_k$ and $H_k$
                2.2.1. Compute average plane ($\pi$)
                2.2.2. Subtract block from its average plane ($\pi$)
                2.2.3. Compress block (2D transform and threshold)
        2.3. Store final sparse structures and $\pi$ coefficients in XDR format
```

Solution Phase

```
1. (If not already in memory) Load compressed blocks
2. Compute force vector
3. Initialise iterative solver with an initial guess
4. While (stop condition not reached)
        4.1. Initialise a residual vector with the computed force vector
        4.2. Apply virtual assembly technique to split the solution vector $x$ into a pair ($x_h$, $x_g$)
        4.3. For each BEM matrix (G,H) - and respective ($x_g$, $x_h$)
                4.3.2. Forward-transform the solution vector $x$ in strips
                4.3.2. For each block do
                        4.3.2.1. Multiply the compressed block by the respective strip
                        4.3.2.2. Inverse-transform the resulting strip
                        4.3.2.3. Add the contribution of the average plane
                        4.3.2.4. Accumulate the result in the residual vector
        4.4. Compute a new solution vector based on the new residual vector
```

Figure 9 Code workflow of the block wavelet transform using virtual assembly technique (BWT-VA)

6. Numerical application

To demonstrate the applicability of the present theory, the problem of a concrete column subject to a heat flux – dependent on the exterior temperature – in each side is analysed, as shown in Figure 10. At the boundaries, the heat flux (q) is proportional to the temperature difference between the column (u) and the ambient temperature (u_s). This boundary condition is expressed by

$$q = -h(u - u_S) \tag{26}$$

where h is the heat transfer coefficient between the column and the air.

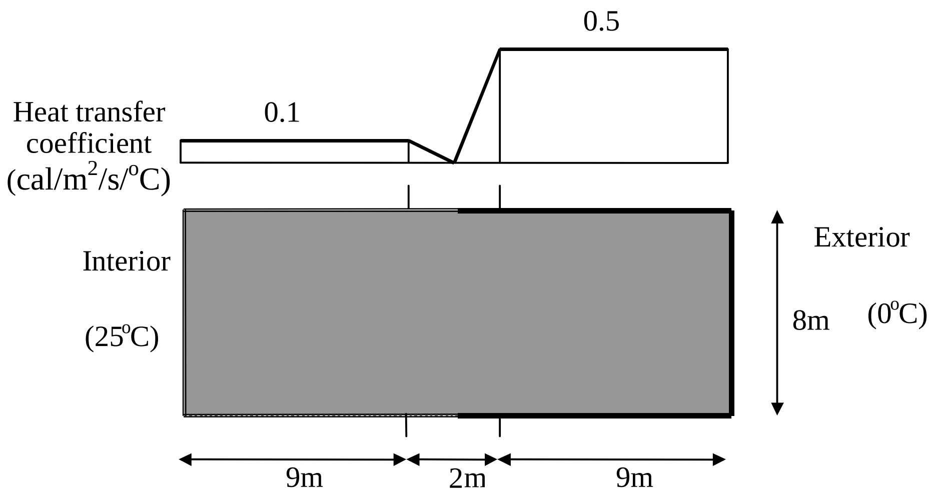


Figure 10 Geometry and boundary conditions, concrete column problem

The interior ambient has a temperature of 25°C and, in this region, the column received a surface treatment to lower its heat transfer coefficient to 0.1 cal/m^2/s/°C. The external ambient has a constant temperature of 0°C and its surface is subject to strong convection, which brings its heat transfer coefficient to 0.5 cal/m^2/s/°C. The value of the heat transfer coefficient decreases linearly to zero in the central region.

Boundary conditions as given by (26) can be represented in the virtual assembly technique as

$$\begin{aligned} \lambda^H x = x, \quad u_0 = 0 \\ \lambda^G x = -hx, \quad q_0 = hu_s \end{aligned} \qquad (27)$$

To analyse the influence of different block sizes, this model was discretized with 4096 constant elements and then solved with both a Gauss solver and the iterative solver implementing the block compression as described in this paper. Although absolute numbers are important, attention must be kept on the behaviour of compression ratio and errors as block sizes are changed.

This model was partitioned using blocks of different sizes: 512, 1024, 2048 and 4096 elements. Each partitioning was submitted to a battery of 400 tests, each one varying thresholds in the range between 10^{-8} to 10^{-3}. Errors were then obtained by comparing the iterative solver solution with the solution obtained with the standard Gauss solver. The wavelet Daubechies with 6 coefficients [8] was used for the necessary transform.

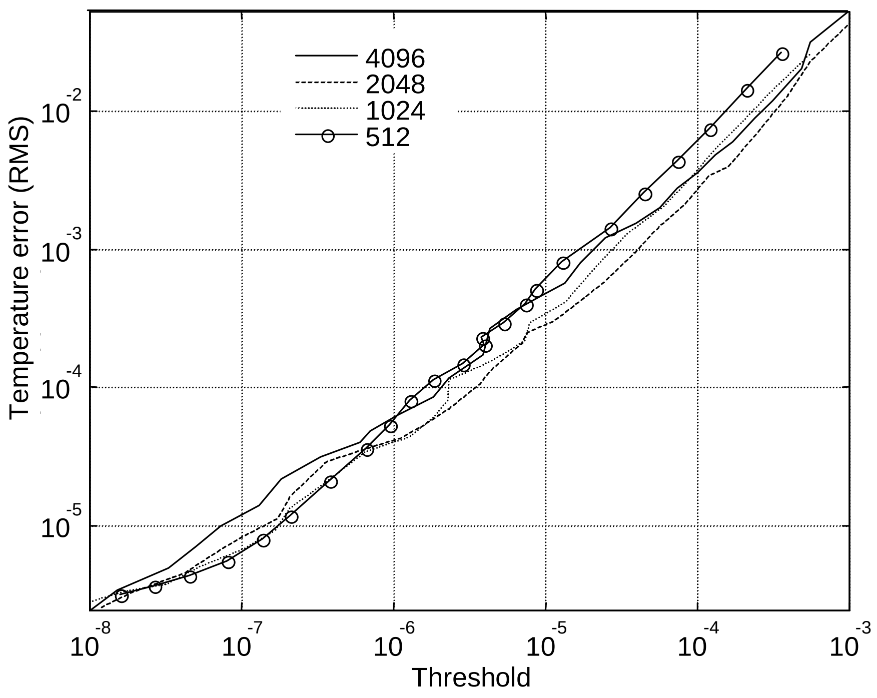


Figure 11 Evolution of the error for the temperature, for different block sizes and threshold values

Figure 11 suggests that the error is not dependent on block size, what is corroborated by successive experiments. As partition grids can be placed anywhere inside the boundary element matrices, it is very difficult to find a theoretical upper bound for the error above, apart from very special cases where the mapping between degrees of freedom in the model and their respective equations inside each block is simple.

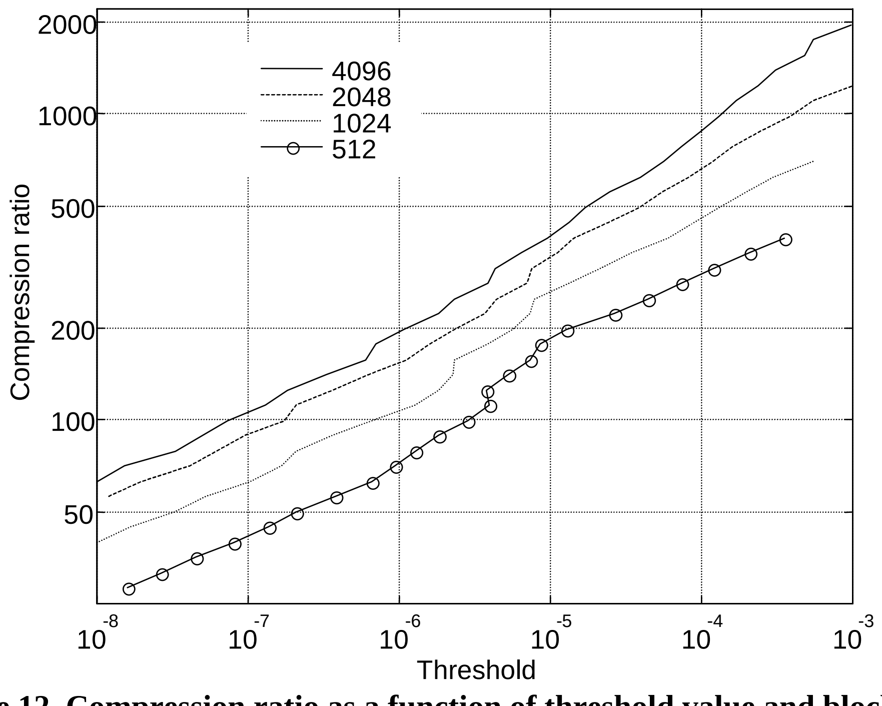


Figure 12 Compression ratio as a function of threshold value and block sizes

Figure 12 shows that there is a direct relation between block sizes and compression ratios. As block size increases, the resulting compression ratio also increases, for the same error.

The above findings simplify the effort of correctly choosing compression parameters since the resulting error is independent of block size. This parameter, therefore, can be chosen independently as a function of the available memory and computer time.

A new BEM model was then created using this same physical model but now discretized using 32,768 constant elements per side, which totals 131,072 degrees of freedom. A Pentium III 900MHz with 1.5GB RAM was used to solve this problem.

In the assembly phase, the ***G*** and ***H*** matrices were partitioned into 256 blocks of identical size – 8192. Following the algorithm in Figure 9, blocks were calculated, compressed and then saved in XDR format. This process generated 256 files that consumed 360 Megabytes of disk space (roughly a single CD), against 262 Gigabytes that would be necessary to store the two uncompressed matrices, which results in a final compression ratio of 720 to 1. The times to compress and store each block can be depicted in Table 2, where respective times for blocks with 2048 and 4096 elements were also included for comparison purposes.

Table 3 Times (in seconds) to generate and compress blocks with sizes 2048, 4096 and 8192 elements

Block size (elements)	**2048**	**4096**	**8192**
Block size (in memory)	64 MB	256 MB	1 GB
Integration	6.2	27.2	126.7
Interpolation plane	1.0	7.8	37.2
2-D Transform	2.8	21.2	101.4
Sparsification	0.6	2.0	7.0
Storage	0.7	0.9	2.1
Total (per block)	11.3	59.1	274.4

It is noticeable that, unlike the integration and sparsification steps that scale well with block size, the interpolation and transform steps took approximately 25% more time than expected – a four time increase as block size doubles – due to the characteristic $\boldsymbol{O}(N^2 log^2 N)$ behaviour of these algorithms. This increase, especially in the case of the 2D transform, is also due to memory cache; for the case with block size 2048, both matrices ***G*** and ***H*** completely fit into the memory cache of the machine that executed these tests and, therefore, all convolutions were done entirely in fast memory. Blocks of 4096 and 8192 elements, however, can hardly stay in cache due to the competition from other processes, including the operating system.

In the solution phase, combining 10 different external heat transfer coefficients and 10 internal heat transfer coefficients created a set of 100 different load cases. Each load case was solved using the same parameters:

- GMRES iterative solver with diagonal preconditioning;
- Stopping criterion: relative residual RMS error in the force vector less than 10^{-12};
- Restart in 25 iterations.

The diagonal preconditioning feature was only possible because the diagonal elements were stored uncompressed within the compressed blocks where they originally lie.

As these problems were impossible to run with well-known solvers as plain iterative solvers (matrix in uncompressed form) or Gauss elimination, we estimated the error by interpolating the value of a 8192 DOF problem solved with a Gauss solver into the 131,072 elements of the current one, which is not a bad estimate since most of the temperature and heat fluxes variations are smooth. The relative RMS error of the temperature was not allowed to grow above 10^{-3}. Performance results are shown in Table 3.

Table 4 Main statistics of the solution of a 131,072 element problem using the block wavelet transform plus virtual assembly technique

Load time	6.2 s
Average number of iterations per load case	19 iterations
Average time per iteration	1.7 s
Average solver time per load case	33.1 s

As can be seen in Table 3, the loading time of the whole compressed structure was, on average, 6.2 seconds, which is roughly the time it takes to evaluate the integrals of a single 8,192-element block. This low time is mostly due to the adopted XDR binary format since a text format would take at least 6 minutes and 40 seconds to load, according to the previous estimates. Using XDR not only retained the loading speed at low values but also made it possible to use this dataset in several different platforms.

After loading, no significant operation takes place until the GMRES cycles, each of which took, on average, 1.7 seconds to complete. The number of iterations per load case, for the same tolerance value of 10^{-12}, was roughly the same for most load cases, with minima of 8 and maxima of 40 iterations. The total wall clock time spent to calculate all 100 load cases was 56 minutes and 21 seconds.

7. Conclusions

The results of this work indicate that there is a large class of industrial problems that can greatly benefit from the BWT-VA approach, particularly large models with multiple load cases. These problems benefit from the reusability of the compressed matrices provided by the current approach.

The numerical application demonstrated that a 131,072 DOF problem, which would otherwise require 262 Gigabytes of disk storage, can be compressed to fit the size of a single CD and then reused at any time to solve many different load cases with no need for decompression. Very high compression ratios were obtained (720 to 1), making it possible to achieve very fast loading (6.2 seconds) and solution (33.1 seconds) steps for a 131 thousand DOF problem, which was solved in a rather standard desktop computer.

As the VA technique keeps the iterative process in the same mathematical space as the original, uncompressed problem, all existing theoretical upper bounds and estimates may still be used in full. The combined BWT-VA technique takes as input the results of widely available integration algorithms, as those found in most textbooks, and therefore there is no need to develop new quadrature algorithms. The application of boundary conditions, which is a major problem for most similar algorithms, is only considered in a phase posterior to compression.

The block transform theoretically allows the solution of problems of any size by breaking the problem into small blocks that are compressed separately, therefore exchanging memory size by computational time, which is a softer limitation in most cases. Numerical experiments concluded that block sizes directly influence the resulting compression ratio, but do not affect error levels, which depend strictly on the adopted thresholds. Given the potentially high compression ratios, it was shown that the coordinate compressed sparse method is the best alternative, with XDR standard being used for individual element storage, improving loading speed, disk size and portability.

The work also shows that, for very high compression ratios, several blocks can be wiped out by the thresholding operation. To bypass this problem, a plane interpolation scheme was presented that represents these otherwise degenerated blocks while not introducing any overhead in the solver phase.

8. References

[1] Bucher, H.F., Wrobel, L.C., Mansur, W.J. and Magluta, C., Fast solution of problems with multiple load cases by using wavelet-compressed boundary element matrices, *Comm. in Numerical Methods in Engineering*, **19**, 387-399, 2003.

[2] Rokhlin, V., Rapid solution of integral equations of classical potential theory, *J. Comp. Phys.*, **60**, 187-207, 1983.

[3] Hackbusch, W. and Nowak, Z.P., On the fast matrix multiplication in the boundary element method by panel clustering, *Numerische Mathematik*, **54**, 463-491, 1989.

[4] Nabors, K., Korsmeyer, F.T., Leighton, F.T. and White, J., Preconditioned, adaptive, multipole-accelerated iterative methods for three-dimensional potential integral equations of the first kind, *SIAM Journal on Scientific Computing*, **15**, 713-735, 1994.

[5] Lage, C. and Schwab, C., Wavelet Galerkin algorithms for boundary integral equations, *SIAM Journal on Scientific Computing*, **20**, 2195-2222, 1999.

[6] Haar, A., Zur theorie der orthogonalen funktionensysteme, *Mathematische Annalen*, **69**, 331-371, 1910.

[7] Daubechies, I., Orthonormal bases of compactly supported wavelets, *Commun.Pure Appl. Math.*, **41**, 909-996, 1988.

[8] Daubechies, I., *Ten Lectures on Wavelets*, SIAM, Philadelphia, 1992.

[9] Brebbia, C.A., Telles, J.C.F. and Wrobel, L.C., *Boundary Element Techniques*, Springer-Verlag, Berlin, 1984.

[10] Koro, K. and Abe, K., A practical determination strategy of optimal threshold parameter for matrix compression in wavelet BEM, *Int. J. Numer. Meth. Engng.*,.**57**, 169-191, 2002.

[11] Koro, K. and Abe, K., Determination of optimal threshold for matrix compression in wavelet BEM, *BEM XXIII*, WIT Press, Southampton, 2001.

[12] Gonzalez, P., Cabaleiro, J.C. and Pena, T.F., Parallel iterative scheme for solving BEM systems using fast wavelet transforms, *Developments in Engineering Computational Technology*, Civil-Comp Press, Edinburgh, 2000.

[13] Cetin, C.E., Gerek, O.N. and Ulukus, S., Block wavelet transforms for image coding, *IEEE Trans. Circ. and Syst. for Video Tech.,* **3**, 433-435, 1993.

[14] Huh, Y., Hwang, J.J. and Rao, K.R., Block wavelet transform coding of images using classified vector quantization, *IEEE Trans. Circ. and Syst. for Video Tech*., **5**, 63-67, 1995.

[15] Bova, S.W. and Carey, G.F., A distributed memory parallel element-by-element scheme for semiconductor device simulation, *Comput. Methods Appl. Mech. Engrg.*, **181**, 403-423, 2000.

[16] Carey, G.F., *Computational Grids: Generation, Adaptation and Solution Strategies*, Taylor & Francis, Austin, 1997.

[17] Golub, G.H. and Van Loan, C.F., *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 1989.

[18] Saad, Y., *Iterative Methods for Sparse Linear Systems*, PWS Publishing, New York, 1996.

[19] Srinivasan, R., *XDR: External Data Representation Standard - RFC 1832*, Sun Microsystems, Inc., 1995.