On the Implementation of Boundary Element Engineering Codes on the Cell Broadband Engine

Manoel T.F. Cunha, J.C.F. Telles, and Alvaro L.G.A. Coutinho

Federal University of Rio de Janeiro
Civil Engineering Department / COPPE
P.O. Box 68506 - 21941-972 - Rio de Janeiro - RJ - Brazil
{manoel,telles}@coc.ufrj.br,alvaro@nacad.ufrj.br
http://www.coc.ufrj.br

Abstract. Originally developed by the consortium Sony-Toshiba-IBM for the Playstation 3 game console, the Cell Broadband Engine processor has been increasingly used in a much wider range of applications like HDTV sets and multimedia devices. Conforming the new Cell Broadband Engine Architecture that extends the PowerPC architecture, this processor can deliver high computational power embedding nine cores in a single chip: one general purpose PowerPC core and eight vector cores optimized for compute-intensive tasks. The processor's performance is enhanced by single-instruction-multiple-data (SIMD) instructions that allow to execute up to four floating-point operations in one clock cycle. This multi-level parallel environment is highly suited to applications processing data streams: encryption/decryption, multimedia, image and signal processing, among others. This paper discusses the use of Cell BE to solve engineering problems and the practical aspects of the implementation of numerical method codes in this new architecture. To demonstrate the Cell BE programming techniques and the efficient porting of existing scalar algorithms to run on a multi-level parallel processor, the authors present the techniques applied to a well-known program for the solution of two dimensional elastostatic problems with the Boundary Element Method. The programming guidelines provided here may also be extended to other numerical methods. Numerical experiments show the effectiveness of the proposed approach.

Key words: Cell Broadband Engine, Boundary Element Method, Boundary Elements, Parallel Programming, Vectorization, SIMD

1 Introduction

Limitations on power and memory use and processor frequency are leading hardware manufacturers to develop new architectures that are changing the programming paradigms established in the last decades. The performance of a large number of existing serial codes no longer benefits from the rising multi-core technology without a proper porting to these environments. Even parallel applications may need some redesign and rewriting to obtain optimum performance on contemporary microprocessors. One clear example are vectorization techniques much used in the past with vector computers that are now surpassed by the SIMD instructions used in multimedia applications. This particular kind of vectorization differs from old vectorization techniques since it relies on hardware features and extended instruction sets only present on modern processors.

The Cell Broadband Engine is a new architecture that is already playing a significant role in the computing industry in some specific areas [13–15] and the knowledge of its strengths and also its current limitations is a decisive factor for engineers and scientists willing to find high-performance solutions to the increasing complexity of their problems and applications. To achieve this goal, this paper introduces the Cell Broadband Engine Architecture and describes in some detail the porting of a well-known serial engineering code to an efficient multi-level parallel implementation.

The implementation of engineering codes, specially using numerical methods to solve elastostatic problems, usually consists in assembling and solving linear equations systems. Even more sophisticated analysis involving elastoplastics or dynamics can be decomposed into a set of such procedures. To generate these systems of equations, numerical methods like finite or boundary elements compute a number of small matrices that are assembled into the equations system accordingly to boundary conditions defined by the problem. In our application, these 2x2 floating-point arrays are specially suited to be computed with SIMD instructions and the paper describes in detail the use of such instructions and how the original algorithm is rewritten to benefit from this vectorization approach.

The text also shows how the proposed algorithm takes advantage of the parallel nature of the Boundary Element Method to efficiently distribute the generation of the equations system among the multiple cores. Since each of the eight computing cores addresses only 256 KB of memory, another challenge to the implementation of engineering codes is the efficient division of the problem - data and code - to fit the memory restraints of these cores. Here, the authors describe the memory transfer mechanisms available on the Cell BE and introduces the use of advanced techniques to hide communication latencies.

The present text is organized as follows: the section 2 presents an outline of the boundary element theory and the following section describes the selected application. Section 4 introduces the Cell Broadband Architecture, its memory transfers mechanisms and the Streaming SIMD Extensions while Section 5 details the multi-level parallel implementation of the code. In section 6 a performance analysis is presented. The paper ends with a summary of the main conclusions.

2 Outline of the Boundary Element Method

The Boundary Element Method (BEM) is a technique for the numerical solution of partial differential equations with initial and boundary conditions [1].

Using a weighted residual formulation, Green's third identity, Betty's reciprocal theorem or some other procedure, an equivalent integral equation can be obtained and converted to a form that involves only surface integrals performed over the boundary. The bounding surface is then divided into elements and the original integrals over the boundary are simply the sum of the integrations over each element, resulting in a reduced dense and non-symmetric system of linear equations.

The discretization process involves selecting nodes on the boundary, where unknown values are considered. Interpolation functions relate such nodes to the approximated displacements and tractions distributions on the respective boundary elements. The simplest case places a node in the center of each element and defines an interpolation function that is constant over the entire element. For linear 2-D elements, nodes are placed at, or near, the end of each element and the interpolation function is a linear combination of the two nodal values. High-order elements, quadratic or cubic, can be used to better represent curved boundaries using three and four nodes, respectively.

Once the boundary solution has been obtained, interior point results can be computed directly from the basic integral equation in a post-processing routine.

2.1 Differential Equation

Elastostatic problems are governed by the well-known Navier equilibrium equation which, using the so-called Cartesian tensor notation, may be written for a domain Ω in the form :

$$G\, u_{j,kk} + \frac{G}{1-2\,\nu}\, u_{k,kj} + b_j = 0 \qquad in \Omega \tag{1}$$

subject to the boundary conditions :

$$\begin{aligned} u &= \bar{u} \qquad on\ \Gamma_1 \qquad and \\ p &= \bar{p} \qquad on\ \Gamma_2 \end{aligned} \tag{2}$$

where u are displacements, p are surface tractions, $\bar{u}$ and $\bar{p}$ are prescribed values and the total boundary of the body is $\Gamma = \Gamma_1 + \Gamma_2$. G is the shear modulus, ν is Poisson's ratio and b_j is the body force component. Notice that the subdivision of Γ into two parts is conceptual, i.e., the same physical point of Γ can have the two types of boundary conditions in different directions.

2.2 Integral Equation

An integral equation, equivalent to Eqs. (1) and (2), can be obtained through a weighted residual formulation or Betty's reciprocal theorem. This equation, also known as Somigliana's identity for displacements, can be written as :

$$u_i(\xi) = \int_\Gamma u^*_{ij}(\xi, x)\; p_j(x)\; d\Gamma(x) - \int_\Gamma p^*_{ij}(\xi, x)\; u_j(x)\; d\Gamma(x) \tag{3}$$

where $b_i = 0$ was assumed for simplicity and the starred tensors, u^*_{ij} and p^*_{ij}, represent the displacement and traction components in the direction j at the field point x due a unit load applied at the source point ξ in i direction.

In order to obtain an integral equation involving only variables on the boundary, one can take the limit of Eq. (3) as the point ξ tends to the boundary Γ. This limit has to be carefully taken since the boundary integrals become singular at ξ. The resulting equation is :

$$c_{ij}(\xi)\; u_j(\xi) + \int_\Gamma p^*_{ij}(\xi, x)\; u_j(x)\; d\Gamma(x) = \int_\Gamma u^*_{ij}(\xi, x)\; p_j(x)\; d\Gamma(x) \tag{4}$$

where the coefficient c_{ij} is a function of the geometry of Γ at the point ξ and the integral on the left is to be computed in a Cauchy principal value sense.

2.3 Discretization

Assuming that the boundary Γ is discretized into N elements, Eq. (4) can be written in the form :

$$c_{ij}\; u_j + \sum_{k=1}^{N} \int_{\Gamma_k} p^*_{ij}\; u_j\; d\Gamma = \sum_{k=1}^{N} \int_{\Gamma_k} u^*_{ij}\; p_j\; d\Gamma \tag{5}$$

The substitution of displacements and tractions by element approximated interpolation functions in Eq. (5) leads to :

$$\mathbf{c_i}\; \mathbf{u_i} + \sum_{\mathbf{k=1}}^{\mathbf{N}} \mathbf{h}\; \mathbf{u} = \sum_{\mathbf{k=1}}^{\mathbf{N}} \mathbf{g}\; \mathbf{p} \tag{6}$$

which can be rearranged in a simpler matrix form :

$$\mathbf{H}\; \mathbf{u} = \mathbf{G}\; \mathbf{p} \tag{7}$$

By applying the prescribed boundary conditions, the problem unknowns can be grouped on the left-hand side of Eq. (7) to obtain a system of equations ready to be solved by standard methods.

This system of linear equations can be written as :

$$\mathbf{A}\; \mathbf{x} = \mathbf{f} \tag{8}$$

where $\mathbf{A}$ is a dense square matrix, vector $\mathbf{x}$ contains the unknown tractions and displacements nodal values and vector $\mathbf{f}$ is formed by the product of the prescribed boundary conditions by the corresponding columns of matrices $\mathbf{H}$ and $\mathbf{G}$. Note that Eq. (8) can be assembled directly from the elements $\mathbf{h}$ and $\mathbf{g}$ without need to generate first Eq. (7).

2.4 Internal Points

Since Somigliana's identity provides a continuous representation of displacements at any point $\xi \in \Omega$, it can also be used to generate the internal stresses. The discretization process, described above, can also be applied now in a post-processing routine.

3 The Application Program

The program reviewed here is a well-known Fortran code presented by Telles [1] for the solution of two dimensional elastostatic problems using linear elements.

The main program defines some general variables and arrays, integer and real, as shown below :

```
program MAIN

  integer :: NN,NE,NP,IPL,INFB,NT,NN2,info
  integer,parameter         :: NMAX=4000
  integer,dimension(NMAX)   :: IDUP
  integer,dimension(NMAX*2) :: IFIP,ipiv
  integer,dimension(NMAX,2) :: INC

  real :: E,PO,C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11
  real,dimension(NMAX)          :: X,Y,C
  real,dimension(NMAX*2)        :: P,XM
  real,dimension(NMAX*2,NMAX*2) :: A

  ! data input
  call INPUT
  ! compute matrix A and independent term XM
  call MATRX
  ! solve system of equations
  call SGESV ( NN2, 1, A, NMAX*2, ipiv, XM, NMAX*2, info )
  if ( info == 0 ) then
    ! output results
    call OUTPT
  else
    write (*,*) 'SGESV : ',info
  endif

end program MAIN
```

The `INPUT` routine reads the program data, the `MATRX` routine computes matrix **A** and the right hand side vector **f**, stored in vector **XM**, while the `OUTPT` routine prints the boundary solution, computes and prints boundary stresses and internal displacements and stresses. The original `SLNPD` subroutine is here replaced by the LAPACK solver `SGESV` [2] which is being ported for the Cell BE processor [8, 9].

Subroutine `MATRX` generates the system of equations by assembling directly matrix **A** without creating the global **H** and **G** matrices. This is done by considering the prescribed boundary conditions for the node under consideration before assembling. The leading diagonal submatrices corresponding to **H** are calculated using rigid body translations. Consequently, when the boundary is unbounded a different type of rigid body consideration needs to be applied.

The element influence coefficients are computed calling subroutine `FUNC`. This routine computes all the element integrals required for the system of equations, internal displacements and internal stresses. Numerical integrals are performed over non-singular elements by using Gauss integration. For elements with the singularity at one of its extremities the required integrals are computed analytically to obtain more accurate results.

The boundary stresses are evaluated using subroutine `FENC` that employs the interpolated displacements and tractions to this end. Here, the contribution of adjacent elements to the common boundary nodes is automatically averaged for non-double nodes. The internal displacements and stresses are obtained by integrating over the boundary elements using subroutine `FUNC`.

The solver is usually the most time consuming routine in BEM programs and various studies have been published on this matter [2]. However, the generation of the equations system as well as the computing of internal points together can take the most part of the processing time [5] and demand special care. While many high-performance parallel solvers are available from standard libraries [2], those two procedures are usually implemented by the researcher and greatly limit the speedup if not optimized. Hence, the Cell BE programming techniques are here applied to the generation of the system of equations and the evaluation of internal point displacements and stresses can also be implemented with the same techniques.

4 The Cell Broadband Engine

The Cell Broadband Engine is a new architecture that succeeds the well-known PowerPC architecture. The Cell BE processor joins in a single chip one PowerPC Processor Element (PPU) and eight Synergistic Processor Elements (SPU). While the PPU runs the operating system and usually the control thread of an application, the SPUs are independent processors optimized to execute data-intensive routines and threads.

At the time of this writing, software for Cell BE is written with C/C++ compilers with vector/SIMD multimedia extensions. However, different SIMD instructions sets for the PPU and SPUs force the programmer to compile separated objects (code modules) in a Cell BE application. Indeed, in a high-level language source code, the SIMD intrinsics for the SPEs are not the same for the PPE which are also different from the PowerPC Altivec instructions, even when executing exactly the same operation.

4.1 The Cell BE Memory Model

The Cell BE Architecture implements a radically new memory organization where PPEs and SPEs access memory in different ways. While the PPE accesses the whole system address space, the SPEs can only address its own private memory. Direct memory access (DMA) commands are used to move data between the main memory and the local memory of each SPEs. With no cache or other hardware mechanisms to automatically load code and data when needed, this memory model leaves to the programmer the task of scheduling DMA transfers between the PPE and the eight SPEs efficiently.

Each SPE private memory includes a 256 KB local storage (LS) to be shared by code and data and 128 registers 128-bits wide. One load and store unit handles data transfers between the local storage and the register file while asynchronous DMA transfers are supported by the Memory Flow Controller (MFC). The MFC supports a maximum transfer size of 16 KB and peak performance is achieved when both the effective (main memory) and LS addresses are 128-bytes aligned and the transfer size is an even multiple of 128.

Besides DMA, Mailboxes is another primary communication mechanism used to exchange queued 32-bits messages. Mailboxes are an useful way to transfer memory addresses and general counters from the PPE to SPEs and can also be used by SPEs to notify the PPE that a memory transfer or computational task has ended.

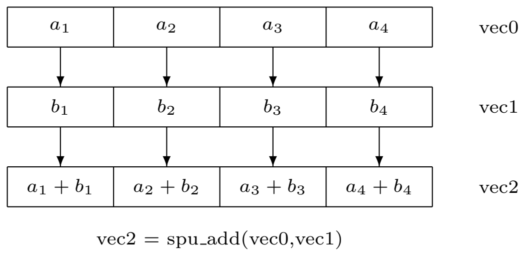
A third type of communication mechanism, signal notification registers, will not be addressed here. More details on the Cell Broadband Engine Architecture can be found in the literature [10–12].

4.2 The Vector/SIMD Multimedia Instructions

Computers were originally classified by Flynn's taxonomy according to instructions and data streams as SISD (single-instruction single-data), SIMD (single-instruction multiple-data), MISD (multiple-instruction single-data) and MIMD (multiple-instruction multiple-data) [6].

As the name suggests, the SIMD model applies to systems where a single instruction processes a vector data set, instead of scalar operands and SIMD instructions perform one operation on two sets of four floating-point single-precision values, simultaneously, as illustrated in Figure 1.

Fig. 1. SIMD Addition



Cell BE provides a large set of SIMD operations. For a full description of all SIMD intrinsic functions the reader is referred to [10]. The implementation of the code here in study with SIMD instructions will be addressed in the next section.

5 The Cell BE Implementation

In the application under study, an equation system is generated in routine `MATRX` with its influence coefficients computed by subroutine `FUNC`. This routine evaluates all the non-singular element integrals using Gauss integration. For elements with the singularity at one of its extremities the required integrals are computed analytically. In the first case, a set of small matrix operations are initially computed, as follows :

$$\begin{bmatrix} UL_{11} & UL_{12} \\ UL_{21} & UL_{22} \end{bmatrix} = -C1 \left[\begin{bmatrix} C2\ logR & 0 \\ 0 & C2\ logR \end{bmatrix} - \begin{bmatrix} DR_{11} & DR_{12} \\ DR_{21} & DR_{22} \end{bmatrix} \right]$$

$$\begin{bmatrix} PL_{11} & PL_{12} \\ PL_{21} & PL_{22} \end{bmatrix} = -C3 \left[\left[\begin{bmatrix} C4 & 0 \\ 0 & C4 \end{bmatrix} + 2 \begin{bmatrix} DR_{11} & DR_{12} \\ DR_{21} & DR_{22} \end{bmatrix} \right] DRDN + C4 \begin{bmatrix} 0 & DRBN_{12} \\ DRBN_{21} & 0 \end{bmatrix} \right] \frac{1}{R}$$

Those 2x2 matrices can be converted into vectors of size 4 and matrix operations can be performed with vector instructions. Thus, a straightforward approach is to use SIMD to evaluate those matrices leaving some intermediate operations to be executed with scalar instructions.

In the original algorithm, those matrices are computed from 2 to 6 times, accordingly to the number of Gauss integration points defined by the chosen integration rule. Alternatively, a fully vector implementation of the matrix computation above can be achieved by using 4 Gauss integration points and evaluating all four values of each coefficient at once, including the intermediate values.

In the application under observation, for each integration point i, the matrix coefficients can be computed as :

$$\begin{aligned}
XMXI_i &= CTE_i * DXY1 + XXS \\
YMYI_i &= CTE_i * DXY2 + YYS \\
R_i &= \sqrt{XMXI_i^2 + YMYI_i^2} \\
DR1_i &= XMXI_i \ / \ R_i \\
DR2_i &= YMYI_i \ / \ R_i \\
UL11_i &= DR1_i^2 - C2 * \log R_i \\
UL22_i &= DR2_i^2 - C2 * \log R_i \\
UL12_i &= DR1_i * DR2_i \\
DRDN_i &= DR1_i * BN1_i + DR2_i * BN2_i \\
PL11_i &= (C4 + 2 * DR1_i^2) * DRDN_i \ / \ R_i \\
PL22_i &= (C4 + 2 * DR2_i^2) * DRDN_i \ / \ R_i \\
PL12_i &= (2 * DR1_i * DR2_i * DRDN_i + C4 * (DR2_i * BN1_i - DR1_i * BN2_i)) \ / \ R_i \\
PL21_i &= (2 * DR1_i * DR2_i * DRDN_i - C4 * (DR2_i * BN1_i - DR1_i * BN2_i)) \ / \ R_i
\end{aligned}$$

Initially using two-dimensional arrays and executed with scalar instructions, the computation presented above - including the intermediate operations - are now performed on vectors and four values are evaluated in each operation. Most of those operations can be performed with basic memory and arithmetic SIMD instructions introduced in the previous section. An SSE implementation of the vector computation being discussed is presented in Listing 1.

For each integration point i, **UL** and **PL** are used to compute two other matrices, **G** and **H** :

$$\begin{bmatrix} G_{11} & G_{12} & G_{13} & G_{14} \\ G_{21} & G_{22} & G_{23} & G_{24} \end{bmatrix} = \begin{bmatrix} G_{11} & G_{12} & G_{23} & G_{24} \\ G_{21} & G_{22} & G_{23} & G_{24} \end{bmatrix} + \left[\begin{bmatrix} UL_{11}^i & UL_{12}^i \\ UL_{21}^i & UL_{22}^i \end{bmatrix} * B_1^i \begin{bmatrix} UL_{11}^i & UL_{12}^i \\ UL_{21}^i & UL_{22}^i \end{bmatrix} * B_2^i \right] * W^i$$

$$\begin{bmatrix} H_{11} & H_{12} & H_{13} & H_{14} \\ H_{21} & H_{22} & H_{23} & H_{24} \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} & H_{23} & H_{24} \\ H_{21} & H_{22} & H_{23} & H_{24} \end{bmatrix} + \left[\begin{bmatrix} PL_{11}^i & PL_{12}^i \\ PL_{21}^i & PL_{22}^i \end{bmatrix} * B_1^i \begin{bmatrix} PL_{11}^i & PL_{12}^i \\ PL_{21}^i & PL_{22}^i \end{bmatrix} * B_2^i \right] * W^i$$

Listing 1.

```
DXY1  = spu_splats(DXY[0]);        //             DXY1
DXY2  = spu_splats(DXY[1]);        //             DXY2
tmp0  = spu_splats(xxs);           //                    X[II] - XS
tmp1  = spu_splats(yys);           //                    Y[II] - YS
XMXI  = spu_madd(CTE,DXY1,tmp0);   // .5 (XI + 1) DXY1 + X[II] - XS
YMYI  = spu_madd(CTE,DXY2,tmp1);   // .5 (XI + 1) DXY2 + Y[II] - YS
tmp2  = spu_mul(YMYI,YMYI);        //               YMYI^2
tmp3  = spu_madd(XMXI,XMXI,tmp2);  //      XMXI^2 + YMYI^2
INVR  = rsqrtf4(tmp3);             // sqrt(XMXI^2 + YMYI^2)
DR1   = spu_mul(XMXI,INVR);        // XMXI / R
DR2   = spu_mul(YMYI,INVR);        // YMYI / R
LOGR  = logf4(INVR);               //                  log R
BN2   = spu_splats(BN[1]);         //                   BN2
UL12  = spu_mul(DR1,DR2);          //    DR1 DR2
DR11  = spu_mul(DR1,DR1);          //    DR1^2
DR22  = spu_mul(DR2,DR2);          //    DR2^2
tmp4  = spu_mul(DR2,BN2);          //           DR2 BN2
tmp5  = spu_mul(DR1,BN2);          //                                 DR1 BN2
BN1   = spu_splats(BN[0]);         //       BN1
UL11  = spu_madd(C2v,LOGR,DR11);   // DR1^2 + C2 log R
UL22  = spu_madd(C2v,LOGR,DR22);   // DR2^2 + C2 log R
tmp6  = spu_madd(DR11,TWO,C4v);    //  2 DR1^2 + C4
tmp7  = spu_madd(DR22,TWO,C4v);    //  2 DR2^2 + C4
tmp8  = spu_add(UL12,UL12);        //  2 DR1 DR2
tmp9  = spu_msub(DR2,BN1,tmp5);    //                       DR2 BN1 - DR1 BN2
DRDN  = spu_madd(DR1,BN1,tmp4);    // DR1 BN1 + DR2 BN2
tmp10 = spu_mul(tmp6,DRDN);        // (2 DR1^2 + C4) DRDN
tmp11 = spu_mul(tmp7,DRDN);        // (2 DR2^2 + C4) DRDN
tmp12 = spu_mul(tmp9,C4v);         //                   C4 (DR2 BN1 - DR1 BN2)
PL11  = spu_mul(tmp10,INVR);       // (2 DR1^2 + C4) DRDN / R
PL22  = spu_mul(tmp11,INVR);       // (2 DR2^2 + C4) DRDN / R
tmp13 = spu_msub(tmp8,DRDN,tmp12); //  2 DR1 DR2 DRDN - C4 (DR2 BN1 - DR1 BN2)
tmp14 = spu_madd(tmp8,DRDN,tmp12); //  2 DR1 DR2 DRDN + C4 (DR1 BN2 - DR2 BN1)
PL21 = spu_mul(tmp13,INVR);        // (2 DR1 DR2 DRDN - C4 (DR1 BN2 - DR2 BN1)) / R
PL12 = spu_mul(tmp14,INVR);        // (2 DR1 DR2 DRDN + C4 (DR1 BN2 - DR2 BN1)) / R
```

Each one of the 2x4 matrices above can be splitted into two 2x2 matrices, as sampled below :

$$\begin{bmatrix} G_{11} & G_{12} \\ G_{21} & G_{22} \end{bmatrix} = \begin{bmatrix} G_{11} & G_{12} \\ G_{21} & G_{22} \end{bmatrix} + \begin{bmatrix} UL_{11}^i & UL_{12}^i \\ UL_{21}^i & UL_{22}^i \end{bmatrix} * B_1^i * W^i$$

Since all values of **UL** are stored in vectors, it is quite simple to perform the multiplications of each value by the respective four values stored in B_1 and W. However, there is no SIMD instruction to perform the sum of the elements of a vector needed in the computation of **G**. Using the SIMD shuffle instructions, the values stored on four vectors can be reordered to obtain the same effect of a matrix transposition, although here the operations are performed on vectors. A possible SIMD implementation of the computations just presented is presented in Listing 2.

Well-known optimization techniques usually applied to scalar codes can also be used in the implementation of vector algorithms in order to replace long latency instructions and to reduce data dependence. Data dependence is the major obstacle to the vectorization of any algorithm. Even well-written programs en-

Listing 2.

```
vector unsigned char permvec1 = {0,1,2,3,16,17,18,19,4,5,6,7,20,21,22,23};
vector unsigned char permvec2 = {8,9,10,11,24,25,26,27,12,13,14,15,28,29,30,31};
vector unsigned char permvec3 = {0,1,2,3,4,5,6,7,16,17,18,19,20,21,22,23};
vector unsigned char permvec4 = {8,9,10,11,12,13,14,15,24,25,26,27,28,29,30,31};
...
tmp0  = spu_mul(UL11,B1W);
tmp1  = spu_mul(UL12,B1W);
tmp2  = spu_mul(UL22,B1W);
tmp3  = spu_shuffle(tmp0,tmp1,permvec1);
tmp4  = spu_shuffle(tmp0,tmp1,permvec2);
tmp5  = spu_shuffle(tmp1,tmp2,permvec1);
tmp6  = spu_shuffle(tmp1,tmp2,permvec2);
tmp7  = spu_shuffle(tmp3,tmp5,permvec3);
tmp8  = spu_shuffle(tmp3,tmp5,permvec4);
tmp9  = spu_shuffle(tmp4,tmp6,permvec3);
tmp10 = spu_shuffle(tmp4,tmp6,permvec4);
tmp11 = spu_add(tmp7,tmp8);
tmp12 = spu_add(tmp9,tmp10);
tmp13 = spu_add(tmp11,tmp12);
Cv    = spu_splats(C);
CC1   = spu_mul(Cv,C1v);
G1    = spu_mul(tmp13,CC1);
...
```

close data dependencies due to the nature of the applications. High performance techniques are presented by the authors in previous studies [3, 4] and will not be addressed here.

The boundary element method has a parallel nature since each boundary node generates two rows on the equations systems. The computing of each pair of rows is totally independent and can safely be performed concurrently. Hence, a straightforward approach is to distribute the boundary elements equally between the eight SPEs. The same procedure can also be used to the computing of internal points. With no communications needed between the SPEs and taking in account that each boundary node must be integrated with all elements on the boundary, an efficient approach is to leave to the SPEs the task of moving and processing data while the PPE only starts the same thread on each SPE, transferring the global addresses of input data vectors and arrays, as demonstrated in Listing 3.

In each SPU, the SPU number (spu_id) is read from the PPU using mailbox and the input data is transfered from main memory to local arrays using DMA. In this implementation, the boundary nodes are evenly distributed among the SPUs and only a pair of rows of the equations system corresponding to each node is computed and transfered from the local storage to the main memory in each iteration, as shown in Listing 4.

The concurrent computing of a given pair of rows with the asynchronous DMA transfer of the previous pairs of rows is a technique used to hide memory transfer latencies, known as double-buffering.

Due to 256 KB local storage size, the initial approach of loading all the input data in SPU's local arrays limits the number of nodes to approximately 4000. In an alternative implementation, only parts of each input array can be transfered

Listing 3.

```
typedef struct {
  float *X;
  ...
} BESTRUCT;
float X[NMAX] __attribute__((aligned(128)));
...
BESTRUCT bestruct __attribute__((aligned(128)));
bestruct.X = X;
...
extern spe_program_handle_t bizep_spu;
int main(void) {
  ppu_pthread_data_t ppdata[8];
  for (i=0;i<NSPU;i++) {
    ppdata[i].context = spe_context_create(0,NULL);
    spe_program_load(ppdata[i].context,&bizep_spu);
    ppdata[i].entry = SPE_DEFAULT_ENTRY;
    ppdata[i].argp = (void *) &bestruct;
    ppdata[i].envp = (void *) 128;
    pthread_create(&ppdata[i].pthread,NULL,&ppu_pthread_function,&ppdata[i]);
    spe_in_mbox_write(ppdata[i].context,&i,1,SPE_MBOX_ANY_NONBLOCKING);
  }
  for (i=0;i<NSPU;i++) {
    pthread_join(ppdata[i].pthread,NULL);
    spe_context_destroy(ppdata[i].context);
  }
  printf ("End of PPU thread\n");
  return 0;
}
```

Listing 4.

```
// the MATRX routine is now the main function running on the SPE
int main(unsigned long long speid,unsigned long long argp,unsigned long long envp) {
  BESTRUCT bestruct __attribute__((aligned(128)));
  // read the SPU id using mailbox
  unsigned int spu_id = spu_read_in_mbox();
  // transfer the structure data from PPU to SPU using DMA
  int tag = 1, tag_mask = 1<<tag;
  mfc_get(&bestruct,(unsigned int) argp,envp,tag,0,0);
  mfc_write_tag_mask(tag_mask);
  mfc_read_tag_status_all();
  // transfer vectors and arrays using DMA
  mfc_get((char *) X,(unsigned long int) bestruct.X,16384,tag,0,0);
  mfc_read_tag_status_all();
  ...
  for (i=first_node;i<=last_node;i++) { // loop over boundary nodes
    for (j=1;j<=bestruct.NE;j++) { // loop over boundary elements
      FUNC(ICOD,C,II,IF,XS,YS,G,H); // SIMD routine
      ...
    }
    // transfer local array A using DMA
    ppu_address = (unsigned long int) (i-1) * sizeof(A);
    for (j=0;j<4;j++)
      mfc_put((char *) A+j*16384,(unsigned long int) ppu_address+j*16384,16384,tag,0,0);
    mfc_read_tag_status_all();
  }
  return 0;
}
```

from the PPU to the SPU. Although increasing the number of DMA transfers, this technique reduces the memory size demand and increases the maximum number of the nodes to be processed. Using 1 GB main memory (QS20), the total number of nodes is limited to 6000 while in a 2 GB system (QS21) it is limited to 10000, approximately.

6 Results

The parallel implementation presented here run on a Cell Blade QS21 server with two processors and 2 GB main memory shared between the processors. Each 3.2 GHz processor has a 64-bits PowerPC with two 32 KB L1 caches and a 512 KB L2 cache and eight SPUs with 256 KB memory each. The operating system is Linux Fedora 7 with Cell BE SDK 3.0.

The study case to be presented here corresponds to a square plate under biaxial load, as found in [1]. The schematic description of the problem is depicted in Figure 2.

Figure 2. A square plate under biaxial load

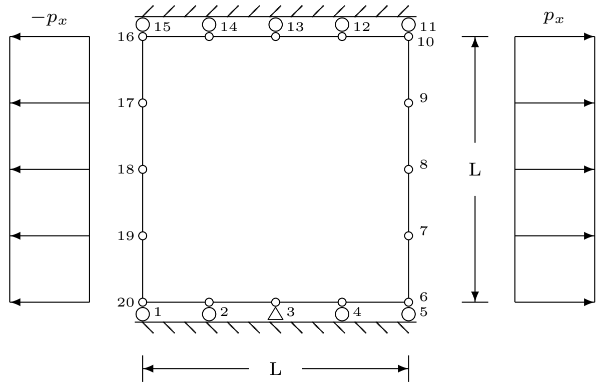


Table 1. QS21 Results - 4000 nodes

SPUs	1	2	4	8
real	7.617s	3.822s	1.926s	0.978s
user	0.002s	0.002s	0.002s	0.002s
sys	0.148s	0.151s	0.159s	0.171s
Speedup	-	1.993	3.955	7.788

The results shown on Table 1 refer to the generation of a 8000x8000 equations system while Table 2 refers to a 20000x20000 equations system of single-precision

Table 2. QS21 Results - 10000 nodes

SPUs	1	2	4	8
real	3m4.297s	1m32.350s	46.252s	23.245s
user	0.002s	0.002s	0.002s	0.003s
sys	0.887s	0.901s	0.920s	1.098s
Speedup	-	1.996	3.985	7.928

Table 3. Intel Xeon Results - 4000 nodes

time	Original	Autovectorization	SSE Intrinsics
real	5.132	3.604	1.992
user	0.212	0.152	0.196
sys	0:5.35s	0:3.75	0:2.18

floating-point elements . The use of a smaller number of SPUs is presented here only for sake of reference, since in practice there is no sense to leave a vector core idle. Also for sake of reference, the results of the SIMD implementation [6, 7] of the same code on another architecture, a quadcore Intel Xeon 2.66 GHz (X5355) processor with 8 GB memory, is presented on Tables 3 and 4.

The almost linear speedups shown in Tables 1 and 2 show the effectiveness of the algorithm used here and emphasize the parallel nature of the Boundary Element Method. The technique of distributing the boundary nodes between the SPUs can also be used to distribute workload between the cores of a blade and among multiple blades. The same approach is used in the shared and distributed memory implementations of this and other BEM codes [4, 5] and will not be discussed here.

It must be noticed that the results shown in Table 1 refers to an implementation where all the input data are loaded to the SPE local store while Table 2 refers to an implementation where only parts of input data are transfered during the runtime, as explained in the previous section. In the first case, most of DMA transfers (99%) are used to write the equations system into main memory. In the second case, to bypass the SPE local store size limitation, most of DMA transfers (92%) are performed to load the input data into SPE's local store. A radical change in the input data layout could reduce DMA reads and will be implemented in a subsequent work.

Table 4. Intel Xeon Results - 10000 nodes

time	Original	Autovectorization	SSE Intrinsics
real	45.822	32.918	12.880
user	1.184	0.956	1.160
sys	0:47.02	0:33.88	0:14.06

7 Conclusions

The Cell Broadband Engine processor is a new architecture developed originally to be used in game consoles and multimedia devices. To face the current limitations on power and memory use and processor frequency, the Cell Broadband Engine introduces a multi-core processor with a highly innovative memory model. As one of the many options of a changing industry, this paper addresses the viability of this environment to run engineering codes, specially numerical methods applications.

Here, the basic aspects of Cell BE architecture and its programming techniques are presented with the porting of a well-known boundary element code to solve two-dimensional elastostatic problems. As shown, existing codes can be rewritten to run on Cell BE after a careful change of the serial algorithm in order to benefit from the multiple vector cores. The results presented here show the effectiveness of the proposed algorithm and emphasize the parallel nature of Boundary Elements. The same parallelization technique can be used to distribute the workload between the SPUs, the cores of a Cell BE blade or among multiple blades. At the time of this writing, these results clearly show the Cell BE well suited to run the kind of engineering application presented here.

However, some current limitations of Cell BE must be taken in account. The first implementation of this family of processors is designed to handle efficiently single-precision floating-point operations while double-precision are usually ten times slower. With no cache and other hardware mechanism developed to handle the processor-memory performance gap, the Cell BE leaves to the programmer the task of scheduling data transfers between main memory and local storages efficiently. This radical design leads to greater learning and programming efforts. A very limited amount of memory in each vector core also implies in significant changes on existing algorithms resulting in increasing development costs and loss of portability.

With the implementation of efficient double-precision floating-point operations, larger memory, a greater number of vector cores and a set of development tools, the next generations of Cell Broadband Engine will play a major role in the computer industry in the near future and become one of the main options for engineering and scientific applications.

Acknowledgments. The authors are in debt to Dr. Michael Perrone, Dr. Ulisses Mello and IBM T.J. Watson Research Center for the support in this work. Hardware resources were provided by IBM Innovation Center, Dallas. M.T.F. Cunha is supported by a CAPES grant from the Ministry of Education, Brazil.

References

1. Brebbia CA, Telles JCF, Wrobel LC. Boundary Elements Techniques : Theory and Applications in Engineering. Berlin: Springer Verlag; 1984.

2. Dongarra J et al. LAPACK Users Guide. 3rd ed. SIAM; 1999.
3. Cunha MTF, Telles JCF, Coutinho ALGA. On the Parallelization of Boundary Element Codes Using Standard and Portable Libraries. Engineering Analysis with Boundary Elements. 2004. 28/7:893-902. doi: 10.1016/j.enganabound.2004.02.002
4. Cunha MTF, Telles JCF, Coutinho ALGA. A Portable Implementation of a Boundary Element Elastostatic Code for Shared and Distributed Memory Systems. Advances in Engineering Software. 2004. 37/7:893-902. doi: 10.1016/j.advengsoft.2004.05.007
5. Cunha MTF, Telles JCF, Coutinho ALGA. Parallel Boundary Elements : A Portable 3-D Elastostatic Implementation for Shared Memory Systems. Lecture Notes in Computer Science. 2005. 3402:514-526.
6. Cunha MTF, Telles JCF, Ribeiro FLB. Streaming SIMD Extensions Applied to Boundary Element Codes. Advances in Engineering Software. 2008. doi: 10.1016/j.advengsoft.2008.01.003
7. Cunha MTF, Telles JCF. On The Vectorization of Engineering Codes Using Multimedia Instructions. Engineering Analysis with Boundary Elements. 2008. *under revision.*
8. Kurzak J, Buttari A, Dongarra J. Solving Systems of Linear Equations on the Cell Processor Using Cholesky Factorization. LAPACK Working Note 184, CS-UTK Tech Report 07-596.
9. Kurzak J, Dongarra J. Implementation of the Mixed Precision in Solving Systems of Linear Equations on the Cell Processor. LAPACK Working Note 177, CS-UTK Tech Report 06-580. Concurrency: Practice and Experience. 2007. 19/10:1371–1385.
10. Cell Broadband Engine Programming Tutorial. IBM. 2007.
11. Cell Broadband Engine Programming Handbook. IBM. 2007.
12. C/C++ Language Extensions for Cell Broadband Engine Architecture. IBM. 2007.
13. Liu Y, Jones H, Vaidya S et al. Speech Recognition Systems on the Cell Broadband Engine Processor. IBM Journal of Research and Development. 2007. 51/5:583–0591.
14. Kachelrieb M., Knaup M., Bockenbach O.: Hyperfast Parallel-beam and Cone-beam Backprojection Using the Cell General Purpose Hardware. Medical Physics. 2007. 34/4:1474–1486.
15. Bockenbach O., Mangin M., Schuberth S.: Real Time Adaptive Filtering for Digital X-ray Applications. Lecture Notes in Computer Science. 2006. 4190:578–587.