**2nd International Conference on Engineering Optimization**
September 6 - 9, 2010, Lisbon, Portugal

# Towards a Blank Shape Optimization: Enhancement of Procedure and FE Solver

**A.R.S. Correia[1], J.L. Alves[2], M.C. Oliveira[1], R. Padmanabhan[1], L.F. Menezes[1]**

[1]CEMUC, Department of Mechanical Engineering, University of Coimbra, Polo II – Pinhal de Marrocos 3030-788, Coimbra, Portugal, {a.correia,Padmanabhan,marta.oliveira,luis.menezes}@dem.uc.pt
[2] Department of Mechanical Engineering, University of Minho, Campus de Azurém, 4800-058, Guimarães, Portugal, jlalves@dem.uminho.pt

**Abstract**
Sheet metal forming is a challenging technological process widely used by automotive carmakers. The success of a forming operation strongly depends on a large number of parameters. The initial blank shape is one of the most important process parameter amongst them, with direct impact on both the quality of the finished part and its cost. However, due to the problem's physical and geometrical non-linearities, namely, materials' mechanical behavior, contact with friction and tools' geometry, there is no simple procedure to determine the initial optimized blank shape geometry, for a given final geometry.
In this work a method that allows to determine the optimal blank shape for a formed part within few steps, using the deformation behavior predicted by finite element (FE) simulations and salient features of NURBS surfaces, is presented. However, since it is based on FE simulations, the time efficiency of this procedure is strongly dependent on the numerical efficiency of the FE solver. DD3IMP is the in-house static implicit FE code used in this work. The aim of the present work is to show the improvements and effectiveness carried out on the blank shape optimization algorithm, resulting from the enhancements introduced in the FE solver in order to speed-up the main algorithm. The results show that an important increase of the computational efficiency of the optimization process was achieved. Therefore, the improvement of the FE solver will contribute to a wider use of implicit codes for solving large scale forming problems, as well as to use them in inverse analysis and iterative optimization procedures.
**Keywords:** shape optimization, FEM, linear sparse systems, solvers, OpenMP.

## 1. Introduction
Sheet metal forming is a complex deformation process controlled by parameters such as the blank shape, tool geometry, sheet thickness, blank holding force, friction, etc. The initial blank shape is one of the important process parameter that has a direct impact on the quality of the finished part, as well as on the final cost of the formed part. Thus, over the last years many blank design approaches have been proposed to determine the optimum initial blank shape (e.g. [1-4]).
This work presents a method to determine the optimal blank shape for a formed part. In brief, by using an iterative process based on FE simulations and a cost function to evaluate and weight the scattering between the target and the current geometry of the blank, the initial guess will be iteratively corrected until the optimized initial blank shape geometry is achieved. Using the proposed procedure it is possible to predict an optimal blank shape within a few steps. Each step involves FE simulation of the forming process. Thus, the time efficiency of this procedure depends directly on the numerical efficiency of the FE code. Static implicit codes are well known for their robustness, reliability and accuracy of numerical results. However, these codes are also usually known to be very time-expensive. In this work the static implicit in-house code DD3IMP (contraction of Deep Drawing 3D Implicit code) was used.
In order to improve the numerical efficiency of this FE code, different enhancement strategies were followed, namely, (i) state-of-art libraries, (ii) compiler options and (iii) OpenMP directives. The main FE code enhancements will be detailed, namely the implementation of a new direct sparse solver of systems of equations, the implementation of OpenMP directives to parallelize several branches of the main algorithm, and some re-written subroutines that allow a significant local improvement of the computational efficiency. In the following section the proposed procedure to determine the optimal blank shape for a formed part is briefly described and some results are presented. In the third section the enhancements strategies adopted are discussed and the improvements achieved are quantified using a benchmark example. The last section presents the main conclusions.

## 2. Blank shape optimization procedure
The method developed to determine the optimal blank shape for a deep drawn part combines FE analysis with a push/pull technique applied to the peripheral nodes in the blank mesh [5]. This procedure was tested using both DD3IMP and ABAQUS implicit codes and the results indicate that it can help to reduce time and cost [6].

The push/pull technique is based on the selection of some nodes in the outer flange of the initial blank mesh. These nodes, with $\mathbf{X}^{init}$ coordinates, change their position due to the draw-in associated to the deep drawing process. The draw-in can be predicted performing the numerical simulation of the forming process, using FE analysis. At the end of the forming process the selected nodes present a final position, $\mathbf{X}^{final}$. The intersection of their trajectory and the required target contour defines an intersection position, named $\mathbf{X}^{inter}$. Based on these coordinates the push/pull technique estimates the new position of the selected nodes, for the next step of the blank shape optimization procedure. The objective of the optimization algorithm is to identify the difference between the existing flange contour and the required target contour and provide a corrective solution that minimizes the difference.

Generally, to accommodate the continuous variation of the nodal coordinates in optimization procedures, a time-consuming remeshing technique is employed. In the proposed method, a regular and uniform mesh with a dimension large enough to accommodate the probable blank shapes is defined as a base mesh for trimming, with a NURBS surface corresponding to the blank outer surface. The trimming operation of the solid finite element base mesh is performed using the in-house code DD3TRIM, an in-house code used to trim 3D solid finite meshes [7]. The initial blank shape can be determined with the aid of empirical formulae. Based on this information, a corresponding NURBS surface is produced, and the base mesh is cut to generate the finite element mesh. The number of control points of this initial NURBS surface defines the number of design variables. The same number of control points will be used in the determination of the new NURBS surfaces, with the push/pull technique, during the optimization procedure.

Figure 1 illustrates the blank shape optimization procedure used in this work. The initial process parameters like, the tools geometry, the mechanical properties of the blank sheet, the friction conditions and the blank holder force are fixed during the optimization procedure. The base mesh is always used to produce the initial and intermediate blank shapes in accordance with the initial NURBS surface and those created by the algorithm, respectively. In each step the base mesh is trimmed by the current NURBS surface, using DD3TRIM program. This minimizes the influence of blank discretization i.e., the FE mesh variability on the optimization procedure. The mesh cut using the NURBS surface is subjected to deep drawing simulation, using DD3IMP. The flange contour of the formed part is compared with the required target contour. If the flange contour is different from target contour, the initial NURBS surface is corrected and a new NURBS surface is produced depending on its deviation as described below. Once the new coordinates of all control points are determined, the corresponding NURBS curve is extruded to produce an intermediate NURBS surface, which is used to generate an intermediate mesh from the base mesh using DD3TRIM. Starting from this intermediate mesh, the new deep drawing simulation is performed with DD3IMP. The procedure is repeated until an optimal blank shape that results in a flange contour with negligible deviation from the target contour is obtained.

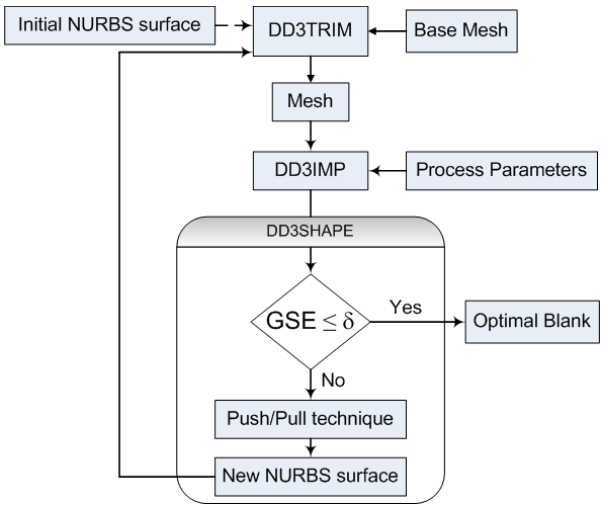


Figure 1. Blank shape optimization procedure.

In order to quantify the deviation between the flange and the target contours, a geometrical measure, known as geometrical shape error, is used. Geometrical shape error ( GSE ) is defined as the root mean square of the shape difference between the target shape and the deformed shape as [4]:

$$\mathrm{GSE} = \sqrt{\sum_{i=1}^{n} \frac{1}{n} \left| \mathbf{X}_i^{\mathrm{inter}} - \mathbf{X}_i^{\mathrm{final}} \right|^2} \ . \tag{1}$$

The distance between $\mathbf{X}^{\mathrm{inter}}$ and $\mathbf{X}^{\mathrm{final}}$ is evaluated at the end of each simulation, *n* being the number of control points used in the initial NURBS surface. The vector norm used corresponds to the Euclidean.
When the GSE reaches a value less than $\delta$, a threshold value predetermined by the user for a required accuracy in the flange shape, the iterative procedure is stopped because the optimal blank shape for the part has already been obtained. The GSE allows correct estimation of the distance between the actual flange contour and the target contour. However, by definition it is not possible with the GSE to evaluate whether the actual flange contour is more inside or outside the target contour. To clearly understand the shape error, a measure called target shape error $(\mathrm{TSE})$ is used to quantify the magnitude of deviation of the flange contour from the required target contour. The target shape error $(\mathrm{TSE})$, expressed in the same dimensions used for point coordinates, is defined as [5]:

$$\mathrm{TSE} = \frac{1}{n} \sum_{i=1}^{n} \left[ \mathrm{sign} \left( 1 - \frac{\left| \mathbf{X}_i^{\mathrm{init}} - \mathbf{X}_i^{\mathrm{final}} \right|}{\left| \mathbf{X}_i^{\mathrm{init}} - \mathbf{X}_i^{\mathrm{inter}} \right|} \right) \left| \mathbf{X}_i^{\mathrm{init}} - \mathbf{X}_i^{\mathrm{final}} \right| \right] \tag{2}$$

where the sign function is used to identify whether a point is inside or outside the target contour. The target shape error is used mainly to analyze the convergence rate towards the solution, since it allows detecting oscillations around the target contour.

2.1 DD3SHAPE in-house code: Push/pull technique applied to NURBS surfaces

At the end of every deep drawing simulation, the closest nodes to the control points of the NURBS surface are identified. Each nodes initial, $\mathbf{X}^{\mathrm{init}}$, and final, $\mathbf{X}^{\mathrm{final}}$, positions define a straight line which approximates the flow path. This straight line intersects the target contour at a point, $\mathbf{X}^{\mathrm{inter}}$ as illustrated in Figure 2. The new set of points to be interpolated is computed using

$$\mathbf{Q}_k = \mathbf{X}_k^{\mathrm{init}} + \xi \left( \mathbf{X}_k^{\mathrm{inter}} - \mathbf{X}_k^{\mathrm{final}} \right) \ \text{with} \ k = 0,...,n-1 \ \text{and} \ \xi \in [0,1] \tag{3}$$

where $\xi$ is a damping factor introduced to smooth the oscillations that may occur between the several NURBS surfaces produced during the iterative procedure. The push/pull technique defines, in each step a *p*th-degree new non-rational B-spline curve *C*, defined by

$$\mathbf{Q}_k = C(\bar{u}_k) = \sum_{i=0}^{n-1} N_{i,p}(\bar{u}_k) \, \mathbf{P}_i \ , \ k = 1,...,n-2 \tag{4}$$

where a parameter value $\bar{u}_k$ is assigned to each $\mathbf{Q}_k$ using, for example, the chord length method and

$$\begin{aligned} N_{i,0}(u) &= \begin{cases} 1 \text{ if } u_i \le u \le u_{i+1} \\ 0 \text{ otherwise} \end{cases} \\ N_{i,p}(u) &= \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u) \end{aligned} \tag{5}$$

$\mathbf{P}$ are the $n+1$ new control points, that are determined using the chord length method and weights equal to 1, solving the $(n+1)\times(n+1)$ system of linear equations in Eq.(5). This curve defines the new blank shape and it is extruded to a NURBS surface to allow its use as trimming domain for a solid finite element mesh. All this operations are performed by DD3SHAPE in-house code [5].

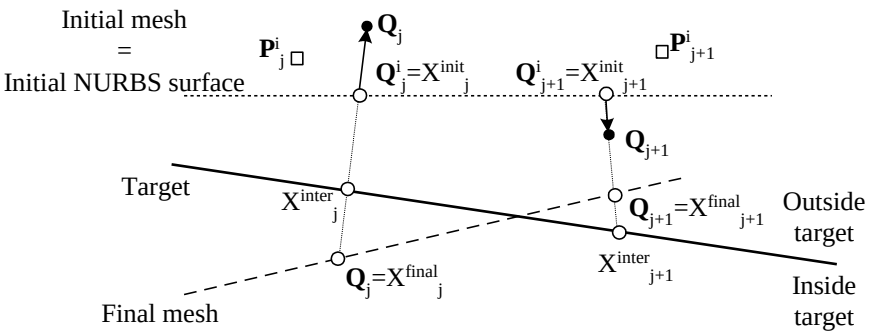


Figure 2. The pull/push technique to create a new curve.

2.2 DD3IMP and DD3TRIM in-house codes

Deep drawing simulations were carried out using the in-house finite element code DD3IMP, developed specifically to simulate sheet metal forming processes [8]. The evolution of the deformation process is described by an updated Lagrangian scheme. An explicit approach is used to calculate an approximate first solution for the nodal displacements, the stress state and frictional contact forces. This first trial solution of the configuration of the deformable body is therefore iteratively corrected. This correction phase is done implicitly using a Newton–Raphson algorithm and finishes when a satisfactory equilibrium of the deformable body is achieved. It is then possible to update the blank sheet configuration at the end of time increment, as well as all the state variables, passing on to the calculation of the next increment until the end of the process. A *r*-min strategy is implemented to impose several restrictions on the size of the time increment in order to improve the convergence. The Coulomb's classical law models the friction contact problem between the rigid bodies (tools) and the deformable body (blank). The contact with friction problem is treated by an augmented Lagrangian approach. The abovementioned fully implicit Newton–Raphson scheme is used to solve, in a single loop, all the non-linearities associated to the problem of contact with friction and the elastoplastic behaviour of the deformable body [8, 9].

In deep drawing processes, the average element size influences results like draw-in prediction, depending upon the complexity of the final shape of the part. In blank shape optimization, it is important to fix this numerical parameter and avoid meshing procedures. In the proposed method, a base mesh is always trimmed to define the initial and intermediate blank shapes, in accordance with the initial NURBS surface or those created by the algorithm. This eliminates the influence of the finite element size in the deformation of the blank. The trimming operation is performed with DD3TRIM, a numerical tool developed to trim solid finite element meshes [7].

2.3 Rectangular cup tool example analysis

In previous works, a rectangular cup geometry was used to demonstrate the robustness of the proposed methodology [6]. Figure 3 shows the target shape and the empirically determined initial blank shape as well as some results concerning the optimization procedure for a mild steel (DC06) blank of 0.8 mm thickness. The initial values for process parameters were chosen based on recommended values and empirical relations (option adopted for the initial blank shape). All numerical simulations were performed with DD3IMP, considering one quarter of the geometries due to symmetries of the part. The base mesh considered an average element size of 1.4 mm with two layers through-thickness, which corresponds to a maximum number of displacement degrees of freedom (d.o.f) of approximately 15000. The base mesh was cut with the initial NURBS surface to produce the initial finite element mesh. The analysis of Figure 3 indicates that, although the initial blank leads to a solution clearly outside the target (Iter 1), the proposed procedure can estimate the optimal blank shape within a few iterations (flange shape error less than 0.5 mm). The number of iterations is influenced by the damping coefficient, as shown in Figure 4, which presents the evolution of TSE and GSE errors, during the optimization procedure. Nevertheless, it is possible to confirm that, typically, a solution can be attained within five iterations.

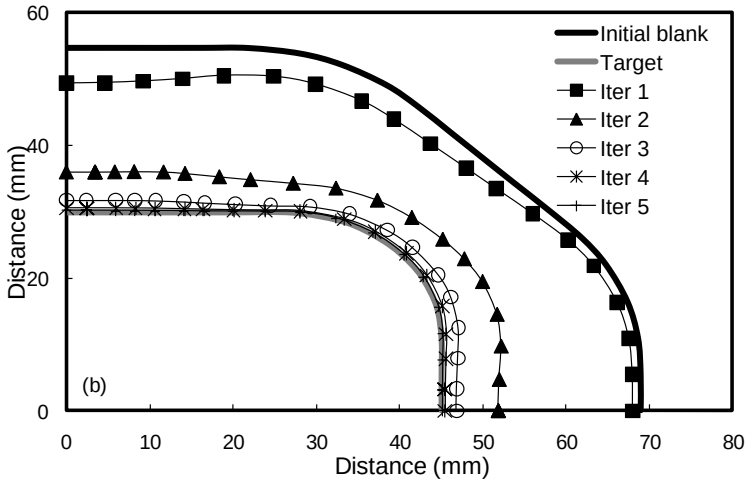


Figure 3. Target shape, initial blank shape and evolution of flange contour over iterations, using a damping coefficient of 0.6 [6].

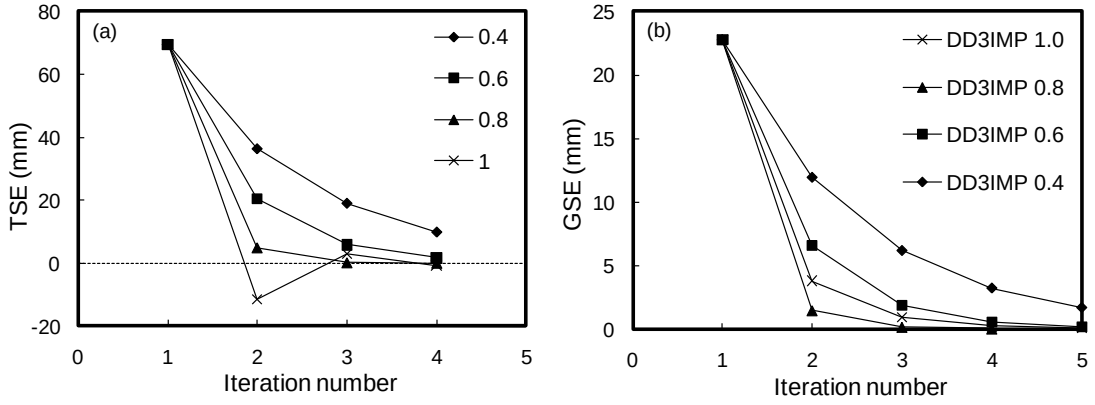


Figure 4. Influence of damping coefficient on (a) TSE (b) GSE [6].

**3. Enhancements strategies**

In the middle of 2009, DD3IMP comprised 1 main program, 315 subroutines, 11 functions, 20 common zones and modules, with about 30000 lines of Fortran 90/95 programming. Following the original structure, defined in the end of the 80s, the variables were divided into two super-vectors, one for integers and another for real variables. Thus, the dynamic allocation performed comprised only the two super-vectors. Although, this strategy has been considered efficient during a long time, it became a bottleneck for improving the efficiency of the code, in particular for exploring parallel programming strategies. Thus, the code was thoroughly reprogrammed and tested in order to perform the dynamic allocation of all variables, using modules. Through the presentation of this work, the initial version of DD3IMP will be named "V45" and the reprogrammed version "O10". The "O10" version of DD3IMP comprises 1 main program, 341 subroutines, 14 functions and 20 modules, with about 30000 lines of Fortran 90/95 programming.

The following sections briefly describe the work performed and the options adopted to improve the computational efficiency of the "O10" version of DD3IMP. The different options were tested using a benchmark example, corresponding to the numerical simulation of a square cup ('Numisheet'93 benchmark test')[10], using a structured regular mesh of 800 3D 8-node trilinear hexahedral finite elements, corresponding to 1323 nodes and 3969 displacement d.o.f. The impact of each option in the overall performance of DD3IMP code was evaluated using the wall time as measure. Each wall time value presented in this work corresponds to the average of three runs. Also, the coefficient of variation was determined so that if it is greater than 15% the simulation session have to be repeated [11]. Nevertheless, it is important to mention that in all cases the coefficient of variation is always less than 15%.

Moreover, it is important to understand the influence of the hardware and operating system in DD3IMP performance. Thus, different combinations of hardware with operation systems were tested and the results are briefly described. Finally, the efficiency gain, according with the size of the problem to be solved, is also evaluated.

3.1 DD3IMP profile

The critical path (the function calling sequence that leads to the bottleneck location) of both versions was identified with the aid of Intel VTune Performance Analyzer for Windows. The critical path identified for both versions is the same, which allowed an extra confirmation of the reprogramming stage performed, and corresponds to the one expected by the development team. The critical path is controlled by the iterative Newton-Raphson procedure (`Itereq` subroutine), being essentially dictated by the computational time necessary to solve the linear sparse systems (`Solve` subroutine) of equations and the assembly of the stiffness matrix and right-hand side vector (`Elemco` subroutine). Besides the critical path identification, Intel VTune Performance Analyzer for Windows also allowed to quantify the time spent in each process. Figure 5 presents the percentage time for the most expensive subroutines, in "V45" version of DD3IMP. The quantification of the critical points in terms of computation time allowed to:

i. Identify the linear sparse systems of equations as critical (~ 58% of the total computation time);
ii. Following the critical path and the caller/callee sequence identify subroutines that should be rewritten;
iii. Identify the branches of the main algorithm, namely the ones related to the loops over the finite elements and contact nodes, for which the implementation of OpenMP directives can contribute to improve their efficiency.

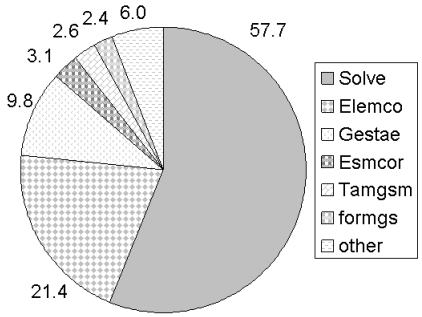


Figure 5. Percentage time for the most expensive subroutines in DD3IMP.

3.2 Rewritten Subroutines

An important aspect that arises from the identification of the critical path and quantification of the average total time per call of each subroutine is the potential gain in performance, if specific parts of the algorithm are rewritten more efficiently. In fact, before exploring other optimization strategies the algorithms implemented in the code should follow programming guidelines that render improved performance. The modular structure of DD3IMP code made this task easier. The caller/callee sequence was analyzed, for each of the critical path subroutines, in order to identify children subroutines with an important contribution for the computational time. This bottom-up (start from callee to caller) strategy allows improving the code algorithm in an organized and structured approach. The subroutines identified based on the profile were mainly connected with rotation operations, between local and global frames and the calculation regarding Bézier surfaces, used to describe the forming tools. Two examples are presented here: (i) `Rotcef`, which rotates the elastoplastic module to the global frame and (ii) `Derpuv`, which deals with the calculation of Bézier surfaces first and second order derivatives. These subroutines were rewritten in order to minimize the arithmetic operations and memory access operations in the `DO` loops, which can involve the need to perform some auxiliary operations in previous `DO` loops.

These improvements were introduced in version "O10" of DD3IMP Code and Table 1 compares the average total time per call of the subroutines before and after the mentioned actions, determined using the previously described benchmark. The table also presents the fold gain time associated to the rewritten subroutines. It is important to mention that these subroutines are called a high number of times during the simulation process, which increases the overall impact of this optimization strategy. As an example, in the benchmark test `Rotcef` is called more than $6\times10^6$ times and `Derpuv` more than $44\times10^6$ times. For the benchmark example selected the reprogramming of these subroutines results in a reduction in the total simulation time of approximately 25%. Thus, these changes were integrated in "O10" version of DD3IMP and the following examples presented always refer to this option.

Table 1. Average total time per call for subroutines in DD3IMP before and after been rewritten.

| Average total time per call [μSeconds] of "O10" version | | | |
|---|---|---|---|
| Subroutine | Before rewriting | After rewriting | Fold gain time |
| `Rotcef` | 11.14 | 3.6 | 3.09 |
| `Derpuv` | 6 | 2.2 | 2.73 |

3.3 Solver

The effectiveness of a specific solver (direct or iterative) or pre-conditioner is, among others, a direct function of the problem dimension and, mainly, of the pattern of non-zeros of the sparse matrix. Thus, to study solvers for linear systems efficiency several aspects should be taken into account: (i) the type of problem to simulate; (ii) the pattern of the sparse matrices (i.e. the organization and positioning of the non-zero values in these matrices); (iii) the matrix dimension and the number of unknowns; and (iv) the type of pre-conditioner and its parameters.

In a previous work those aspects were explored for DD3IMP in-house code and several direct and iterative solvers were tested considering that the matrix is structurally symmetric (i.e. although the matrix is not symmetrical the distribution of non-zero values is symmetric) [12]. At that time the selected solver was the iterative Conjugate Gradient Squared method (CGS) combined with Level-k ILU preconditioner, based on the incomplete factorization LU method [13, 14]. In fact, the CGS method is largely used to solve linear non-symmetric systems. It is systematically referred as presenting a convergence rate twice faster than similar methods (Conjugate Gradient method, Bi-Conjugate Gradient method, for example) with less memory requirements than Minimum Residual methods [15, 16]. Also, the ILU preconditioners are between the most robust and stable developed until today [13].

One of the fundamental strategies to improve computational efficiency is to resort to state-of-art libraries. Thus, the Direct Sparse Solver (DSS) from Intel Math Kernel Library (MKL) emerged as a natural option, since it is known to be a high-performance, robust, memory efficient and easy to use software for solving large sparse symmetric and unsymmetric linear systems of equations on shared memory multiprocessors. For sufficiently large problem sizes, numerical experiments demonstrate that the scalability of the parallel algorithm is nearly independent of the shared-memory multiprocessing architecture and a speedup of up to seven using eight processors has been observed [17].
The DSS solver was integrated in "O10" version of DD3IMP code and Table 2 presents the results for the benchmark example, using both solvers. The results are clearly advantageous for the DSS solver, which leads to a fold gain time of 2.19. These results confirm the importance of guaranteeing state-of-art algorithms.

Table 2. Wall time for "O10" DD3IMP version with CGS and DSS solvers.

| | "O10" CGS | "O10" DSS |
|---|---|---|
| Wall time [seconds] | 1684 | 768 |
| Fold gain time | - | 2.19 |

3.4 Compiler options
All of the results previously presented were obtained using default Intel Fortran compiler options. Another way to improve the computational performance of the code is to explore the set of features available for the compiler. The Intel Fortran compiler supports a variety of options and features that can improve the code performance, through the exploitation of opportunities, in terms of speed up, in different processor's architecture [18] such as: (i) **more aggressive automatic optimizations** (`/O3`), with which the compiler tries to implement in lining intrinsic functions, constant propagation, copy propagation, global register allocation, scalar replacement, etc; (ii) **interprocedural Optimization** (`/Qipo`), which explores dead function elimination, array dimension padding, common block splitting, etc; (iii) **improves precision of floating-point divides** (`/Qprec-div-`), which improves the speed of the computation, by changing floating-point division computations into multiplication by the reciprocal of the denominator; (iv) **targeting specific** (`/Qax`), generating specialized instructions to SSE4.2, SSE4.1, SSSE3, SSE3, SSE2, and SSE architectures; or (v) **generic architectures** (`/arch:IA32`), which indicates that the code is be able to run at any Intel Pentium (or compatible or later processor). These options correspond to the use of `/fast` compiler option, specific for Intel processors, with the advantage of generating a portable code, for different types of processors. The compiler options presented previously correspond to the following command line, for Windows OS:

```
ifort /O3 /Qipo /Qprec-div- /QaxSSE4.1,SSSE3,SSE3 /arch:IA32 *files*.for
```

The benchmark test was performed with this compiler options and the results are labeled with "-S", since this corresponds to a serial code.
The Intel Fortran compiler also presents an **auto-parallelization** (`/Qparallel`) feature, which automatically translates serial portions of the input program into equivalent multithreaded code [18]. This option determines the loops that are good work sharing candidates, performs the dataflow analysis to verify correct and optimized parallel execution, and partitions the data for threaded code generation as needed in programming with OpenMP directives. The OpenMP and auto-parallelization applications provide the performance gains from shared memory on multiprocessor and multi core systems. The use of auto parallelization was added to compiler options presented previously, leading to the following command line, for Windows OS:

```
ifort /O3 /Qipo /Qprec-div- /QaxSSE4.1,SSSE3,SSE3 /arch:IA32 /Qparallel *files*.for
```

Table 3. Wall time for the benchmark example (3969 and 33489 d.o.f.) with default ("O10" DSS), serial ("O10" DSS-S) and auto-parallel ("O10" DSS-A) compiler options.

| | "O10" DSS | | "O10" DSS-S | | "O10" DSS-A | |
|---|---|---|---|---|---|---|
| Nº of d.o.f | 3969 | 33489 | 3969 | 33489 | 3969 | 33489 |
| Wall time [seconds] | 768 | 61751 | 757 | 26906 | 785 | 21212 |
| Fold gain time to previous | - | - | 1.01 | 2.30 | 0.96 | 1.27 |

The benchmark test was performed using this compiler options and the results are labeled with "-A", since this corresponds to an auto-parallel code. Table 3 presents the wall time results obtained with the different compilation options for the benchmark example performed with a coarse and a fine FE mesh. The coarse mesh is the one already described with 3969 displacement d.o.f. The fine mesh presents 7200 3D 8-node trilinear hexahedral finite element mesh, corresponding to 11163 nodes and 33489 displacement d.o.f. In fact, the coarse mesh indicates similar wall times for all compilation options tested. It is only possible to evaluate the importance of the

compilation options selected using the finer mesh, for which it is shown the enormous impact of the compilations options, even for serial code.

3.5 Hardware and operating system

In order to analyze the influence of hardware and operating system (OS), different combinations were tested considering the CPU specifications and OS shown in Table 4, respectively. In terms of workload the hardware component with higher impact is the CPU thus, the different computers were labeled according to the CPU type. The Duo Core computer ("DI") was tested with both 32 bits platform operating systems ("OSL32" and "OSW32"). The Quad Core computer ("QI") was tested only with the 64 bits platform ("OSW64"). All the results presented in the previous sections were obtained in an Intel Duo Core machine (corresponding to "DI" in the Table 4) using the Windows operating system (labeled "OSW32" in the same table).

Table 4. Specifications of CPU and Operating System tested.

| CPU specification | Label |
|---|---|
| Intel Mobile Core 2 Duo T7200 @ 2.00GHz Caches L1=2 x 32 Kbytes and L2=4096 Kbytes | "DI" |
| Intel Core Quad CPU Q9450 @ 2.66GHz Caches L1=4 x 32 Kbytes and L2=12288 Kbytes | "QI" |
| Operating System (OS) | Label |
| Linux Ubuntu 9.1 desktop edition, 32 bits platform | "OSL32" |
| Windows7 ultimate edition, 32 bits platform | "OSW32" |
| Windows XP professional edition, 64 bits platform | "OSW64" |

Table 5 presents the results obtained with Duo and Quad Core Intel processors, using both Linux and Windows OS. The "V45" version of DD3IMP was also tested, using the different configurations, in order to evaluate the gains in the overall combinations. Globally, the fold gain time is not influenced by the type of processor or the operating system. The operating system seems to have a small influence in the wall time. The important conclusion from this study is the fact that the compiler options adopted for DD3IMP seem to work well through different systems (CPU+OS).

Table 5. Wall time for the benchmark example (3969 d.o.f.) with default ("V45", "O10" CGS, "O10" DSS), serial ("O10" DSS-S) and auto-parallel ("O10" DSS-A) compiler options, using different CPU and OS.

| | "DI OSL32" | | | | | "DI OSW32" | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | "V45" | "O10" | | | | "V45" | "O10" | | | |
| | | CGS | DSS | DSS-S | DSS-A | | CGS | DSS | DSS-S | DSS-A |
| Wall time [s] | 1740 | 1749 | 776 | 754 | 779 | 1772 | 1684 | 768 | 757 | 785 |
| Fold gain time to previous | 1.00 | 0.99 | 2.25 | 1.03 | 0.97 | 1.00 | 1.05 | 2.19 | 1.01 | 0.96 |
| Fold gain time to "V45" | 1.00 | 0.99 | 2.24 | 2.31 | 2.23 | 1.00 | 1.05 | 2.31 | 2.34 | 2.26 |

| | "QI OSW64" | | | | |
|---|---|---|---|---|---|
| | "V45" | "O10" | | | |
| | | CGS | DSS | DSS-S | DSS-A |
| Wall time [s] | 1224 | 1145 | 518 | 544 | 547 |
| Fold gain time to previous | 1.00 | 1.07 | 2.21 | 0.95 | 0.99 |
| Fold gain time to "V45" | 1.00 | 1.07 | 2.36 | 2.25 | 2.24 |

3.6 OpenMP

The main algorithm of DD3IMP is divided, in each increment, in a prediction (explicit approach) and a correction (implicit approach) phases. In each of these phases it is necessary to determine the elemental stiffness matrices and right-hand side vectors, perform the necessary corrections due to contact conditions and assemble the global stiffness matrix and right-hand side vector. These operations involve `DO` loops over the finite elements of the deformable body and over nodes in contact with the tools. Also, it is necessary to evaluate the contact conditions for all potential contact nodes. OpenMP directive `DO` was introduced for the most expensive and parallelizable loops, taking into the necessary directives to control private variables and critical regions.

The benchmark example was performed with the "O10" OpenMP version of DD3IMP (labeled "-Op"). Table 6 compares the results for the auto-parallel and the OpenMP versions of DD3IMP code. The results show the advantages of moving forward to code reprogramming with OpenMP directives, even for small size problems, as the one selected to perform this comparison. The operating system seems to have a small influence in the wall time, as already mentioned in the previous section. The results obtained with the OpenMP version are particularly

interesting for Quad Core configurations.

Table 6. Wall time for the benchmark example (3969 d.o.f.) for "O10" DD3IMP parallel versions for different combinations of OS and CPU types.

| | OSL32 | | OSW32 | | OSW64 | |
|---|---|---|---|---|---|---|
| | DI | | DI | | QI | |
| | -A | -Op | -A | -Op | -A | -Op |
| Wall time [s] | 779 | 435 | 785 | 449 | 547 | 193 |
| Fold gain time to previous | - | 1.79 | - | 1.75 | - | 2.83 |
| Fold gain time to "V45" | 2.23 | **4.00** | 2.26 | **3.95** | 2.24 | **6.33** |

3.7 Influence of the problem dimension

The importance of the problem size was already mentioned in section 3.4 and in this section it will be analyzed further, using the two versions of the code: (i) version "V45" compiled with the features that rendered the best performance, labeled "V45-S" to distinguish from the previous; (ii) version "O10-Op", which rendered the best performance results for the benchmark test. This comparison is performed in order to quantify the total gain in performance associated with the rewritten of subroutines, the implementation of the DSS solver and the use of OpenMP directives. Table 7 presents the meshes definitions used in the benchmark example.

Table 7. Problems size definition.

| Label | Nº of elements | Nº of nodes | Nº of displacement d.o.f |
|---|---|---|---|
| 10x10x2 | 200 | 363 | 1089 |
| 20x20x2 | 800 | 1323 | 3969 |
| 30x30x2 | 1800 | 2883 | 8649 |
| 40x40x2 | 3200 | 5043 | 15129 |

Table 8 presents the results for the different combinations of processors and operating systems tested. The results presented in this table are different from the previously presented since three new runs were performed. Globally, the conclusions of section 3.4 are also valid even for smaller problems. The important conclusion from this study is the fact that the fold gain time between the two versions increases with the problem size and then attains a constant value. The "O10-Op" version is particularly efficient for bigger size problems.

Table 8. Wall time for DD3IMP versions: serial ("V45"-S) and OpenMP ("O10-Op") for different problem syzes. The value in brackets corresponds to the fold gain time.

| | "DI OSL32" | | "DI OSW32" | | "QI OSW64" | |
|---|---|---|---|---|---|---|
| | "V45-S" | "O10-Op" | "V45-S" | "O10-Op" | "V45-S" | "O10-Op" |
| Mesh 10x10x2 | 126 | 48 **(2.6)** | 138 | 59 **(2.3)** | 103 | 22 **(4.7)** |
| Mesh 20x20x2 | 1656 | 429 **(3.9)** | 1707 | 466 **(3.7)** | 1200 | 187 **(6.4)** |
| Mesh 30x30x2 | 5654 | 1427 **(4.0)** | 5658 | 1394 **(4.1)** | 3965 | 576 **(6.9)** |
| Mesh 40x40x2 | 13600 | 3309 **(4.1)** | 13906 | 3649 **(3.8)** | 9477 | 1373 **(6.9)** |

**Conclusions**

The present work deals with an industrial problem, namely the blank shape optimization of sheet metal formed part. Two features of this problem are explored, namely the optimization algorithm itself and the computational efficiency of the FE solver.

The present work clearly shows and emphasizes the advantages of an improvement on the computational efficiency of the FE solver, particularly when used in any iterative optimization problem. Firstly, it is clearly shown that all algorithms and its numerical implementation shall be (re-)written in a more efficient way, and that state-of-art libraries shall be adopted when and where possible. The profiling of the code allows to focus this development effort in large codes by identifying the most expensive algorithms an subroutines, for which the numerical efficiency can be deeply improved, either by reprogramming or by implementation of OpenMP directives. Additionally, compilation options, including auto-parallelization shall also be explored and tuned.

In case of DD3IMP FE solver, the exploitation of all these techniques leads to a speed-up time of more than 4 times for a Dual Core Intel machine and more than 6 for a Quad Core machine. The speed-up time is similar for different operative systems running on the same hardware. Moreover, instead of an exponential evolution of the computational wall-time with the size of the problem (measured by the number of d.o.f.), the new version of the FE

solver shows a linear evolution, allowing to carry out numerical simulations more efficiently as well to solve larger problems.
All the above mentioned improvements recently introduced to the FE solver have a large industrial impact, mainly in the optimization procedures based on FE analysis. A blank shape optimization algorithm is presented. For the proposed benchmark (a rectangular cup), a speed-up time of 4 allows unquestionably to determine an optimal blank shape within a working day, even taking into account the pre and post processing phases.

**Acknowledgements**
The authors are grateful to the Portuguese Foundation for Science and Technology (FCT) and to the Institute for Interdisciplinary Research of the University of Coimbra, who financially supported this work.

**References**

[1] T. Kuwabara, W.-H. Si, PC-based blank design system for deep drawing irregularly shaped prismatic shells with arbitrarily shaped flange, *Journal of Materials Processing Technology*, 63, 89–94, 1997.

[2] Y.Q. Guo, J.L. Batoz, H. Naceur, S. Bouabdallah, F. Mercier, O. Barlat, Recent developments on the analysis and optimum design of sheet metal forming parts using a simplified inverse approach. *Computers and Structures*, 78, 133–148, 2000.

[3] J.-Y. Kim, N. Kim, M.-S. Huh, Optimum blank design of na automobile sub-frame. *Journal of Materials Processing Technology*, 101, 31–43, 2000.

[4] S.H. Park, J.W. Yoon, D.Y. Yang, Y.H. Kim, Optimum blank design in sheet metal forming by the deformation path iteration method, *International Journal of Mechanical Sciences*, 41, 1217–1232, 1999.

[5] R. Padmanabhan M.C. Oliveira, A.J. Baptista, J.L. Alves, L.F. Menezes, Blank design for deep drawn parts using parametric NURBS surfaces, *Journal of materials processing technology*, 209, 2402–2411, 2009.

[6] W. Hammami, R. Padmanabhan, M.C. Oliveira, H. BelHadjSalah, J.L. Alves, L.F. Menezes, A deformation based blank design method for formed parts, *International Journal of Mechanics and Materials in Design*, Springer, 5 (4), 303-314, 2009.

[7] A.J. Baptista, J.L. Alves, D.M. Rodrigues, L.F. Menezes, Trimming of 3D solid finite element meshes using parametric surfaces: application to sheet metal forming, *Finite Element Analysis and Design*, Elsevier, 42, 1053–1060, 2006.

[8] L.F. Menezes, C. Teodosiu, Three-dimensional numerical simulation of the deep-drawing process using solid finite elements, *Journal of Materials Processing Technology*, Elsevier, 97 (1–3), 100–106, 2000.

[9] M.C. Oliveira, J.L. Alves, L.F. Menezes, Algorithms and Strategies for Treatment of Large Deformation Frictional Contact in the Numerical Simulation of Deep Drawing Process, *Archives of Computational Methods in Engineering*, 15, 113-162, 2008.

[10] T. Furuhatha, "Experimental results in the square cup deep drawing", *Proceedings of 2nd International Conference on Numerical Simulations of 3D Sheet Metal Forming Processes – Verification of Simulation with Experimental*, A. Makinouchi, E. Nakamachi, E. Oñate, R.H. Wagoner, R.H. (Eds.), 1993.

[11] D. J. Lilja, "Measuring Computer Performance: A Practitioners Guide", Cambridge University Press, New York, 1-239, 2000.

[12] J.L. Alves and L.F. Menezes, "Resolução de sistemas lineares esparsos em simulação 3-D do processo de conformação de chapa: Estudo da influência da numeração da malha, do precondicionador e do método iterativo", Proceedings of *V Congresso de Métodos Numéricos en Ingenieria*, J.M. Goicolea, C. Mota Soares, M. Pastor e G. Bugeda (Eds.), SEMNI Sociedade Española de Métodos Numéricos en Ingeniería, 106, 2002.

[13] E. Chow and Y. Saad. "Experimental study of ILU preconditioners for indefinite matrices", *Journal of Computational and Applied Mathematics*, 86, 387-414, 1997.

[14] Y. Saad and H.A. van der Vorst, "Iterative Solution of Linear Systems in the 20-th Century", *Journal of Computational and Applied Mathematics*, 123, 1-33, 2000.

[15] D.R. Fokkema, G.L.G. Sleijpen, H.A. van der Vorst. "Generalised conjugate gradient squared", *Journal of Computational and Applied Mathematics*, 71, 125-146, 1994.

[16] G. Meurant, "Computer Solution of Large Linear Systems", North-Holland, 1999.

[17] Intel Corporation. "Intel Math Kernel Library: Reference Manual", *Intel's Document* Number: 630813-023US, 1-2630, 2009.

[18] Intel Corporation. "Intel Fortran Compiler: User and Reference Guides", *Intel's Document* Number: 304970-006US, 1-3824, 2009.