PAM-CRASH on the IBM 3090/VF: An integrated environment for crash analysis

by P. Angeleri
D. F. Lozupone
F. Piccolo
J. Clinckemaillie

PAM-CRASH® is an industrial code developed by Engineering Systems International (ESI) S.A. and designed specifically for automotive crashworthiness analysis. We discuss the problems encountered and describe the solutions provided for an efficient migration of the code on the IBM 3090/VF system. Runs on actual test cases have shown a vector/scalar speedup between 2.7 and 3.5. Moreover, we present the program modifications we have introduced in order to exploit parallel processing using the Multitasking Facility of the VS FORTRAN compiler. Performance results for 3090/VF systems, from the Model 200E to the Model 600E, are given. Finally, we describe the restructuring of the graphic processors, PRE-3D and DAISY, to allow an effective use of the IBM 5080 Graphics System capabilities in providing an integrated design environment for crash analysis.

In the automotive industry, crash design is of vital importance because it can significantly increase the safety of vehicle occupants. Historically, many car manufacturers have obtained information on structural behavior under crash conditions by means of experimental tests on prototypes. In the last ten years, however, vehicle crashworthiness simulation has become increasingly important as mathematical models and computer programs have been developed to simulate structural crash.

Crash phenomena present some of the most complex problems faced by structural mechanics. Since such phenomena are both spatial and temporal, they involve not only contacts among several surfaces, but also complex nonlinear behavior characterized by large strains. Consequently, the solution of such problems requires the computational power provided by a supercomputer, in order to achieve results consistent with the time constraints of the industrial processes.

PAM-CRASH® is a finite-element program for structural crash analysis. The program is an industrial code designed for automotive crashworthiness analysis; it was developed and is maintained by Engineering Systems International (ESI) S.A. Rungis-France.

The new IBM version of PAM-CRASH was developed to take advantage of the vector-processing capabilities of the 3090/VF system in providing users of IBM

systems in the automotive industry with an engineering system for the simulation of crash phenomena. The results of the PAM-CRASH migration allow us to show the improvements in performance available to a user of PAM-CRASH on a 3090 system with the Vector Facility feature. The vector version of PAM-CRASH has been further provided with the capability of exploiting parallel processing on 3090/VF multiprocessors.

We discuss the problems encountered and we describe the solutions provided for an efficient migration of the code on the IBM 3090/VF system. We deal particularly with the problems posed by the class of finite-element codes, which use an explicit algorithm to perform the integration of dynamic equations. Emphasis is placed on the description of the nonlinear shell-element algorithm and on the criteria followed in exploiting the vector-processing capabilities of the 3090/VF. Runs on actual test cases show a vector/scalar speedup between 2.7 and 3.5. Moreover, we present the program modifications we have introduced to exploit parallel processing using the Multitasking Facility of the VS FORTRAN compiler. Performance results for 3090/VF systems, from the Model 200E to the Model 600E, are shown.

A finite-element code can be exploited successfully only if it is provided with graphic programs aiding users in the pre- and post-processing phases. PAM-CRASH is interfaced with the PRE-3D preprocessor and the DAISY postprocessor to form an integrated design environment for crash analysis. Use of the IBM GRAPHIGS Library allows PRE-3D and DAISY to run on the IBM 5080 Graphics System. They supply improved interactivity by means of the 5080 local devices and better performance by exploiting the local processing capabilities of the 5080.

PAM-CRASH code characteristics

PAM-CRASH[1] is a three-dimensional Lagrangian explicit finite-element code for analyzing the dynamic response of structures. The code takes both material and geometrical nonlinearities into account, and it has general contact/impact capability. It is especially designed for automotive crashworthiness analysis[2].

Time integration by finite difference solves the problem of acceleration, velocity, and displacement time histories at each discretization point of the structure. There are two possible integration schemes, *implicit* and *explicit*;[3] both reduce the dynamic equilibrium equations to a set of algebraic equations. The explicit method, used by PAM-CRASH, recasts the dynamic equations into a form yielding nodal accelerations, by which the central finite-difference scheme is applied to obtain nodal velocities and displacements. The solution of the final set of equations is trivial for systems whose mass matrix is lumped. The conditional stability of the explicit integration method forces the evaluation of the integration step size by a stability criterion. In the case of PAM-CRASH this is the Courant criterion, based on the propagation time of the sound waves across the smallest mesh element. The advantages offered by unconditionally stable implicit integration schemes, allowing larger time steps, cannot be exploited in crash simulation because of the characteristics of these phenomena. They are characterized by a short duration and highly nonlinear behavior, and to be correctly described they require a time-step size of the order of that needed by an explicit code. Furthermore, the explicit time integration scheme avoids matrix assembly and inversion, the latter being very expensive for large systems; therefore, it significantly reduces virtual memory requirements.

A reduced integration technique with one-point quadrature has been used to evaluate the element forces.

For spatial discretization, the code only uses the simplest finite-element formulation such as four- or three-node bilinear shell elements and corresponding two-node beam elements. Simple linear elements are preferred to higher-order elements because experience has shown that the former are computationally more cost-effective, even though a greater mesh density is required in areas of severe deformation.

The nonlinear shell element is the element most widely used in automotive crash analysis. The PAM-CRASH shell element is a bilinear four-node quadrilateral element, originally developed by Belytschko[4] and based on Reissner–Mindlin plate theory. This

theory differs from the classical Kirchhoff theory in its treatment of shear deformation. In the Kirchhoff hypothesis, the effect of the transverse shear deformation is neglected, and the finite-element approximation requires shape functions with C^1 continuity. In the Reissner–Mindlin theory, the shear deformation is taken into account, and only C^0 continuity is required, with great simplification in the element formulation. A reduced integration technique with one-point quadrature[5] has been used to evaluate the element forces in the context of the explicit time-integration scheme provided by PAM-CRASH. However, reduced integration in a bilinear plate element permits kinematic (zero-energy) modes. These modes, called *hourglassing* modes in finite-difference literature,[6] can destroy the solution because of the introduction of spurious spatial oscillations. An hourglass control scheme has been implemented to avoid that numerical instability phenomenon.

A wide variety of material laws are available to model elastic, nonlinear, and failure conditions. Rigid walls may be defined to provide external impact surfaces, while a slide-line algorithm can prevent penetration of internal structure surfaces that may collide during the crash simulation. Various slide-line interface conditions are available, including sliding, separation, and friction. The user must specify which region of the structure is to be considered for crash purposes by listing the elements which are part of it. An either partially or totally automatic "boxing strategy" is under development, in which the user need no longer list the elements belonging to the sliding surfaces. Two basic types of contact phenomena[7–9] can be represented. The first type involves two surfaces coming into contact, the second, a single surface buckling and coming into contact with itself. Both situations include friction interface behavior and are based on the penalty method, which consists in placing normal interface springs between the nodes belonging to the penetrating surface and those belonging to the contact surface.

Rezone and restart options are provided to allow mesh redefinition during the analysis and to split larger problems into a set of smaller computer runs.

Enabling environment

The new IBM version of the PAM-CRASH program was developed to utilize the vector-processing[10,11] capabilities of the IBM 3090/VF system[12] in providing the automotive industry with an engineering system for the simulation of crash phenomena. The code was written to run efficiently on different supercomputers; it is installed on non-IBM systems in automotive-industry computing centers. Alternatively, a version of the code is currently running on IBM systems, including 3090s, without exploiting the potential benefits of vector and parallel processing that a vector multiprocessor machine such as the 3090/VF can provide. The results of the PAM-CRASH enabling efforts now allow us to show the performance improvement[13] the user can obtain by using the new IBM version of PAM-CRASH on the 3090 system provided with the Vector Facility feature. The vector version of PAM-CRASH provides the additional capability of exploiting parallel processing on IBM multiprocessor systems.

Preliminary feasibility analysis was the first step of the migration process. It was based essentially on an exchange of information and expertise with Engineering Systems International (ESI) S.A., the engineering software house that developed the code and maintains it. That step permitted a first rough estimation both of the resources needed to carry out the vector enabling and of the expected improvement in performance. Once this preliminary phase was completed, an international team was set up for the purpose of developing a vector version of the PAM-CRASH code provided with parallel capability. At any given time, the team comprised three people: the first from ESI, owner of the code; a second from the European Center for Scientific and Engineering Computing (ECSEC), the European IBM organization for Numerically Intensive Computing (NIC); and a third from IBM Japan, whose customers requested the enabling of the code on the 3090/VF system.

The team composition was defined to cover all the aspects involved in the enabling project:

- Physical formulation and equations involved
- Structure of the code and finite-element technology
- Vector and parallel architecture
- System environment and performance

The code was installed on the IBM 3090-200/VF of ECSEC. At that time, the software environment comprised the following:

- MVS/XA 2.1.3 operating system provided with the JES2 2.1.3 subsystem and the DFP 2.1.2
- TSO/E 1.4 provided with the ISPF 2.2 for the terminal interactive environment

- VS/FORTRAN 2.1,[14–16] the vectorizing compiler, provided with the Multitasking Facility (MTF) for parallel processing
- VS FORTRAN Execution Analyzer,[17] a software tool for determining the percentage of time spent in the different sections of the code.

ESI provided the enabling team with some test cases from the automotive industry, since the code is widely used there. These test cases involved the crash analysis either of a complete automobile or of part of it. Since the test cases provide a wide range of applications associated with different phases of the automotive design process, they have allowed the enabling team to estimate performance improvements in the overall design process by exploiting vector and parallel processing.

PAM-CRASH carries out the integration of the dynamic equation using a finite-difference explicit algorithm.

The team's first task was to refine the feasibility analysis in order to better approximate expected improvements in performance, available resources, time required to reach goals, and, finally, procedures.

Vector feasibility analysis

Once the presence of vector content in an application has been ascertained, vector feasibility analysis[18] is a basic step in assembling the elements necessary to carry out a cost evaluation. That analysis allows one to determine whether the price/performance requirements can be met. The vector feasibility analysis was carried out as follows:

1. Definition of objectives and requirements
2. Analysis of the overall program structure, of the logic organization, and of the coding implementation to build the execution flows and to identify the main code sections
3. Evaluation of the computer resource distribution among the code sections identified in the previous step
4. Analysis of the algorithms of those code sections which are responsible for a comparatively large amount of computer resource utilization, and definition of the algorithm characteristics for a possible vector implementation
5. Evaluation of the vector content in the present implementation
6. Comparison of the results of steps 4 and 5 to identify the potential vector content, to locate the inhibitors preventing vectorization, to evaluate the level effort required to promote vectorization, and finally to estimate the algorithm performance improvement
7. Evaluation of the overall effort and estimation of the global performance improvement according to the objectives defined in step 1
8. Definition of time schedule and planning of human resources

The primary objective of a vector migration process is the reduction of processing time by exploiting the processing capability of the Vector Facility feature. The secondary objective, which can also yield benefits, is the exploitation of system resources such as virtual storage and I/O processing.

PAM-CRASH carries out the integration of the dynamic equation using a finite-difference explicit algorithm. Thanks to the characteristics of this algorithm, widely used in structural mechanics, the virtual memory request is highly reduced compared with that of alternative implicit schemes. According to that, the System/370 architecture provides enough virtual memory to make efficient use of the code in standard applications. However, to meet future customer requirements in terms of model sizes, the exploitation of virtual memory over the 16-MB line, allowed by 370/XA architecture, whose present extension is 2 GB, has been included in the objectives.

The PAM-CRASH program produces hundreds of megabytes of data as a result of the analysis. These data can be used as input for the graphic postprocessing program, DAISY, to aid the user in analyzing the results of the simulation, to check whether the design requirements are met, and eventually to generate further modifications.

The I/O content can be considerable, and the time the program spends to perform I/O activity can become a considerable fraction of the total time. An

I/O algorithm which is either not well tuned or designed for a different system architecture can produce a degradation in performance. We decided to include, among our objectives, I/O activity tuning if performance tests revealed any degradation in elapsed-time performance due to the I/O processing.

Many of the widely used engineering codes were written during the 1970s, and the system architecture for which they were designed has evolved extensively during the last twenty years. However, the software houses that develop and maintain those codes have adopted a conservative policy with respect to software upgrading. Consequently, our experience in vector migration has been that we must expect to face preliminary problems in mapping the code architecture to the hardware characteristics to optimize the system resource utilization.

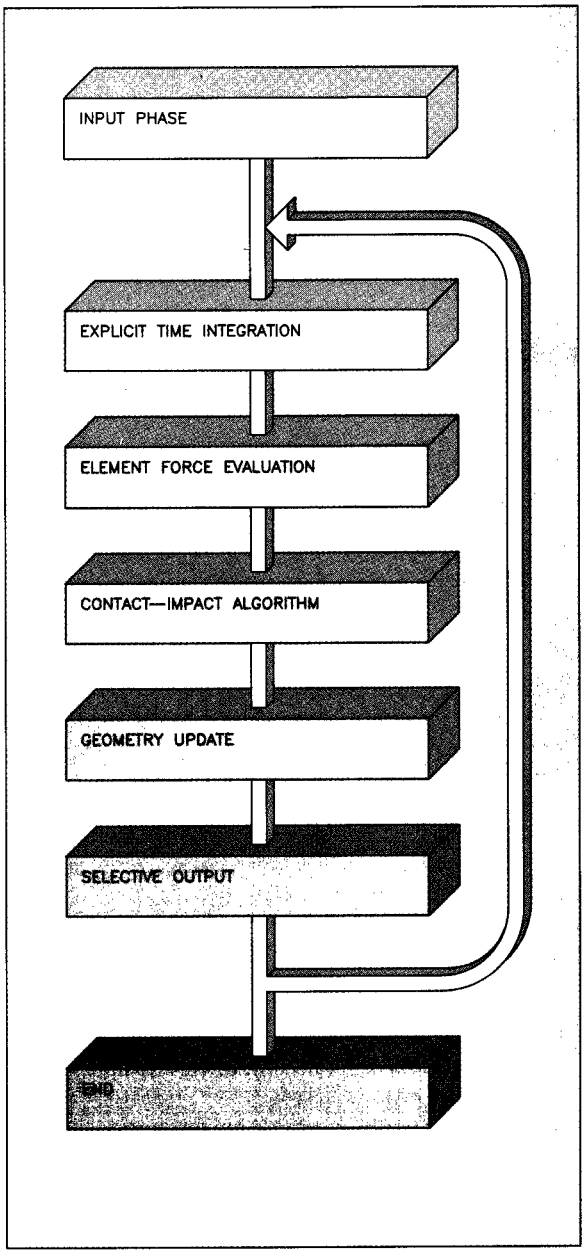
Although the software houses generally deliver different versions of the same programs for use in different environments, they reasonably prefer to limit as much as possible the differences among the versions. Therefore, we had to define a policy regarding the introduction of code changes, taking into account portability, maintenance, and readability requirements.

We have, then, faced the problem of identifying the program structure by analyzing the physical formulation, the algorithms involved, and their relationship to the code implementation. This implies

- Building the logic program organization and locating the main code routines
- Drawing the "tree" of the execution flows and tracing the routine calling sequence
- Identifying the data organization and the process of data transmission

The logic organization and data structure of a finite-element code making use of an explicit algorithm for the time integration are well known in literature[19,20]. However, requirements or programming styles can cause the code implementation of two programs to be very different although they carry out the same tasks using the same algorithms. For example, modularity requirements can affect the mapping of the logical organization of the code sections, and hardware requirements can affect the overall data organization. The flowchart shown in Figure 1 describes the program organization and outlines the main sections.

Figure 1 Program organization and main sections



From preliminary analysis this information has been collected for each code section:

- *Element force evaluation.* Nonlinear shell elements constitute over 90 percent of the elements used.

Table 1 Processing time distribution

Code Section	CPU Percentage
1 Element force evaluation	70–82
2 Geometry update	7–9
3 Explicit time integration	3–4
4 Contact algorithm	17–3
1 + 2 + 3 + 4	97–98

Table 2 Vector percentages compared to total CPU percentage

Code Section	CPU Percentage	Vector Percentage
1 Element force evaluation	70–82	63–74
2 Geometry update	7–9	7–9
3 Explicit time integration	3–4	3–4
4 Contact algorithm	17–3	0–0
1 + 2 + 3 + 4	97–98	73–87

- *Selective output.* I/O activity is not negligible, and it affects the elapsed time.
- *Contact algorithm.* Execution time for the treatment of impact phenomena depends on the modeling choices, since the user can select the region to be analyzed for crash purposes.

By performing a hot-spot analysis, we obtained the percentage distribution of processing time among the sections of the code. We carried out this task using the VS FORTRAN Execution Analyzer, which allows the user to obtain detailed timing information at the subroutine and statement levels. The time distribution depends on the type of the analysis, and the user must select the region of the structure to be analyzed for crash purposes.

Table 1 shows the range of variation of processing time distribution among the different code sections for the test cases provided by ESI. Since those cases are a complete set of the applications currently employed in the design process of the automotive industry, that range can reasonably be considered representative of the general class of problems in the industry.

The remaining 2–3 percent of the CPU time is distributed among other code sections whose single incidence is less than 1 percent. The incidence of the *input phase* section is negligible.

We analyzed the algorithms of those code sections which involved the utilization of a comparatively large percentage of the computer resources, and we defined the characteristics for a possible vector implementation.[21] We discovered the following:

- The code sections of the *geometry update* and of the *explicit time integration* are completely vectorizable.
- The *element force evaluation* algorithm is vectorizable, except for the section assembling the element forces into the global force vector. We estimated the percentage of its incidence at about 10 percent of the force execution time of the element force section.
- The *contact algorithm* is vectorizable, but we ascertained from analysis of the present implementation that vector enabling was feasible by just carrying out a complete reorganization of the code.

We can assert that a good estimation of the vectorizable percentage f, defined as the percentage of the total scalar execution time spent on the vectorizable code sections, varies in a range between 73 and 87 percent. Table 2 shows the vectorizable percentage estimated for each section and compares it to the total CPU percentage.

The theoretical model, known as Amdahl's law,[22] estimates the performance improvement of a vectorizable application:

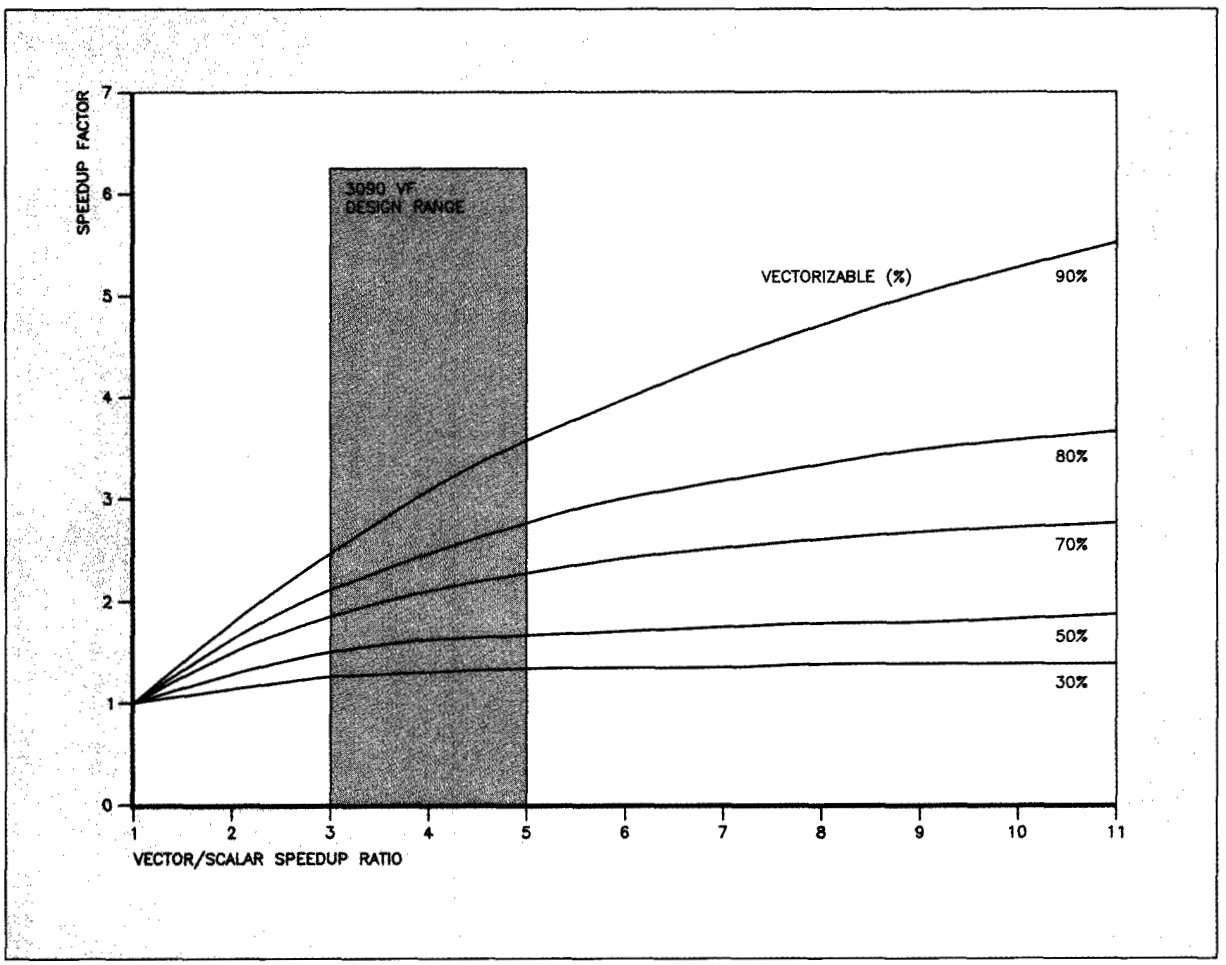
$$S = \frac{1}{(1 - f) + f/sr}, \tag{1}$$

where S is the overall vector/scalar speedup factor, f is the vectorizable percentage, and sr is the vector/scalar speedup obtained by performing the computation of the vectorizable code sections with the Vector Facility.

The vector/scalar speedup design range of the Vector Facility is between 3 and 5. This value is optimized for the middle range of vectorizable percentage. Figure 2 shows the improvement ratio of the execution time of a vector application as a function of the vectorization percentage. Several vector/scalar speedup ratios are given.

The parameter sr depends both on the hardware characteristics of the Vector Facility feature and on the characteristics of the computation performed inside the code. We mention the most important factors affecting the estimation of sr:

Figure 2 Amdahl's law

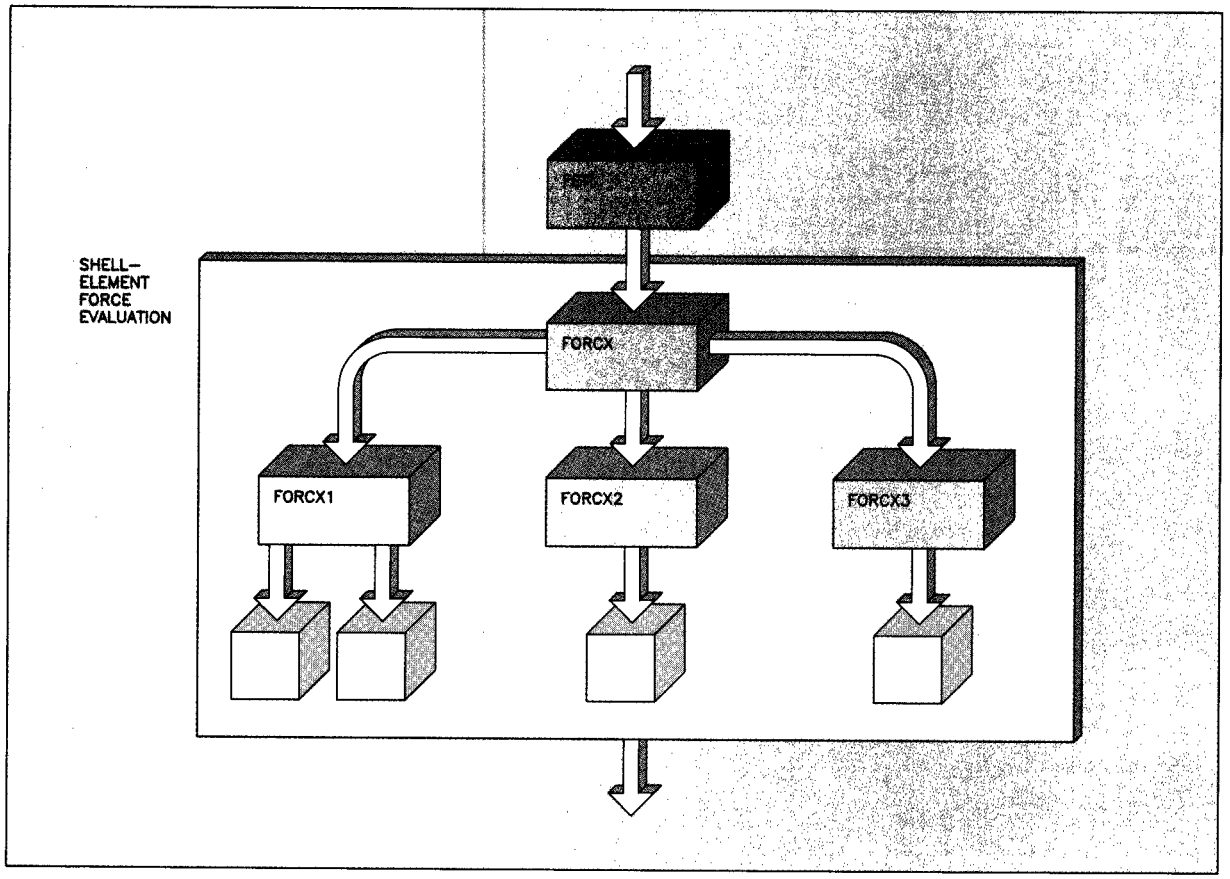


- *Kind of operations involved in the computation.* Different vector operations have different startup times and different speeds in comparison with the scalar instructions.
- *Vector length.* A vector length of 128 elements is close to the optimum speedup. The threshold above which vector code performances are better than scalar ones is approximately 12.
- *Stride of the vector element in storage.* The best vector performances are achieved with the lowest stride, i.e., stride 1. (*Stride* represents the increment used to step through array storage to select the vector elements from the array.) In actual practice, it is important to reduce the stride to minimize cache misses and paging. For particular stride values, the scalar execution is preferred, because it severally results in faster performance than the vector execution.
- *Presence of conditional processing.* An excessive number of IF statements can cause poor performance. When an IF statement is vectorized, the time needed to process the IF statement and the operations governed by it remains the same whether or not the condition is verified.

The operations of the vectorizable code sections are essentially arithmetic in nature. The only functions are the trigonometric functions, sine and cosine, and the square root. The presence of conditional processing is minimal. The data organization is complex and requires many arrays forming the database of the analysis. We have noted that many operations are performed using indirect element selection.

The preceding considerations were formulated to give a correct estimate of the *sr* parameter. However,

Figure 3 Original implementation of the shell algorithm



the *sr* value should also be selected on the basis of practical experience in vector migration. Using these criteria, we selected a value of 4 as a reasonable estimation of *sr*.

By means of Amdahl's law, we have estimated between 2.2 and 2.9 the global speedup *S* of the final enhanced version running in vector mode with respect to the same one running in scalar mode. Yet, we must consider that this speedup takes into account only the performance improvement obtained by exploiting the vector feature. A migration process involves a general code optimization, with a consequent improvement in scalar performances. A good percentage of the efforts required to exploit the Vector Facility involves good programming practices rather than specific vector techniques, yielding a reasonable further improvement in performance of about 20 or at least 10 percent. Consequently, we can expect values for the final global vector/scalar speedup, with respect to the original scalar code, to fall between 2.4 and 3.5.

We have defined procedures for

- Vectorizing the code sections of the *element force evaluation*, of the *geometry update*, and of the *explicit time integration*
- Applying a scalar optimization to the *contact algorithm* section

The global number of statements involved in the migration process is approximately 15 percent of the whole program, of which 5 percent, spread over twelve routines, is associated with vector enabling, while the remaining 10 percent is associated with I/O operations and virtual memory tuning.

The criteria in the modification policy of the original source code have been to limit code changes as much

as possible while addressing the following issues:

- Avoiding imbedding routines in one another
- Applying modifications at routine level only if the performance improvement is over a predefined threshold value
- Preserving the readability of the code

Vector migration process

Considerable effort has been applied in the vectorization of the shell-element force evaluation section because it has a major influence on processing time.

We now describe the restructuring process we have applied to promote vectorization of the code section computing the shell-element forces.[23]

From the initial analysis we realized that the original coding form was developed for implementation on a vector machine with a Cray-like architecture. The original coding structure (Figure 3) comprised the following:

- The main routine, FORCX, carrying out the task of sectioning the shell-element group into subsets of 128 elements. FORCX calls three routines and is called by the driving program routine, FETI, which performs the step-by-step integration of the finite-element dynamic equation.
- Three routines, FORCX1, FORCX2, and FORCX3 computing the shell forces for each subset. They call other small routines to perform particular subtasks.

The processing time spent in FORCX is negligible.

The code was well structured and organized in a DO-loop pattern with a count value of 128, which is one of the optimum values associated with the section size of the vector register. With the aid of the compiler output relative to the vectorization analysis [compiler options: REPORT(XLIST LIST)],[16] we have ascertained that about 50 percent of the DO loops have been correctly vectorized. The remaining 30 percent have presented some inhibitory factor preventing the vectorization. The inhibitors can be classified according to the following different situations:

- Use of intrinsic functions not available in vector mode, e.g., MAX/MIN functions
- The presence of statements inhibiting the vectorization, e.g., CALL and GO TO statements
- Detection by the compiler of recurrence conditions

In the last 20 percent of the possible situations the compiler recognized the DO loops eligible to be vectorized, but vector code was not generated because of performance considerations.

Before approaching the problem of eliminating the inhibitors, we thought it necessary to reformulate the algorithm implementation in order to obtain the best results from the migration process. We felt particularly that the data organization must be redefined, since we could not get the best results from vectorization with the current structure.

The first step in the migration process consisted in restructuring the data organization.

The data of each shell element, once preprocessed in the input phase, are stored in a database comprising several arrays. For the purpose of computing element forces, the program makes use of pointer arrays to locate in the database, wherever necessary, the data associated with each element. Consequently, the program must perform sparse-matrix computation with a large utilization of the indirect element selection.[24] The original developers were not deeply aware of the way in which this kind of data organization could influence the performance of systems consisting of central processor units provided with caches.

The first step of the migration process consisted in restructuring the data organization to localize data references and to promote an efficient utilization of the cache. Our efforts were directed at achieving the following goals:

- Collecting the gather operations in a localized code section at the beginning, before the force computation starts
- Collecting the scatter operations in a localized code section at the end, after the force computation has been completed
- Performing the force computation on contiguous data

- Reducing and optimizing usage of those temporary arrays used to store either gathered data or intermediate results

We have, then, dealt with the problem of eliminating inhibitory factors in order to promote the vectorization of those DO loops for which the compiler does not generate vector code.

The MAX function did not vectorize at the time of the enabling because the VS FORTRAN Compiler Version 2.1 does not support maximum/minimum functions in vector mode. The solution was to replace the function with an IF construct. The problem has since been solved because the most recent VS FORTRAN Compiler Version 2.3 now supports min/max functions.

The FORCX2 routine calls a routine evaluating the element stresses by means of a material relation defined by the user. This routine is small, and we have imbedded it inside the calling routine. In this way we have promoted the vectorization of the DO-loop in whose range the CALL statement was included.

The presence of a GO TO statement in a DO-loop range inhibits vectorization. Although we replaced it with an IF construction to promote vectorization, we did not obtain satisfactory performance results. We then split the original DO loop into two blocks. In the first block, the program computes the number of times the condition for conditional processing is met, and a pointer vector is initialized to identify the array elements on which the computation must be performed:

```
      K=1
      DO 10 I=1,N
      IF(A(I).....)
      K=K+1
      IND(K)=I
      .........
10    CONTINUE
      NK=K
```

In the second block, the operations executed earlier under the control of the IF statement are processed, selecting the array elements by means of the pointer vector evaluated at the 10 DO loop:

```
      DO 20 I=1,NK
      .........
      ....Y(IND(I)). .
      ....Z(IND(I)). .
      .........
20    CONTINUE
```

The 20 DO loop is executed either in vector or in scalar mode depending on the count value. The directive ASSUME COUNT forces the compiler to generate scalar code when the count value is smaller than a threshold.

The recurrence conditions are detected by the compiler when the program performs scattering operations, in order to save element information to be used in the next time step and to assemble the global force vector.

Those operations are performed by means of pointer vectors, previously evaluated in the FORCX, indirectly selecting the array elements on which the computation has to be performed. The compiler cannot recognize whether the recurrence condition involves a backward dependence. Since we know the algorithm, we are sure that no backward recurrences are associated with the former situation. Vectorization has been promoted by means of the directive IGNORE RECRDEPS. In the latter situation, the recurrence condition is detected when the global force vector is assembled. The vectorization cannot be promoted because the assembling algorithm is serial (it involves an unbreakable recurrence condition).

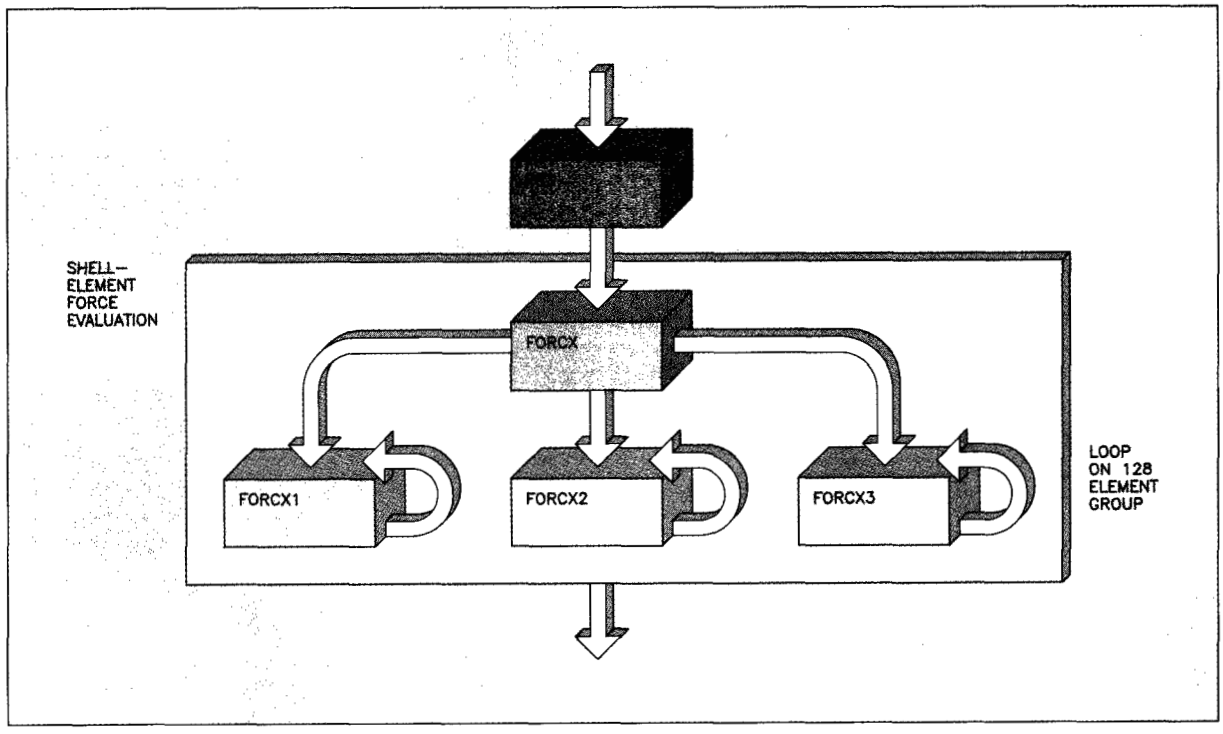
The compiler recognizes 20 percent of the DO loops as eligible for vector analysis, but prefers, for performance reasons, to generate a scalar code. Vectorization has been promoted by means of the directive ASSUME COUNT.

The final coding structure is shown in Figure 4.

We now describe some fine-tuning techniques to optimize the utilization of the vector registers and of the compound instructions which have yielded remarkable improvements in performance.

Vector-register usage can be optimized by keeping in mind that the Vector Facility feature makes available sixteen vector registers in single precision and eight registers in double precision. The compiler uses

Figure 4 Vector implementation of the shell algorithm



the registers to store the intermediate results of the operations in the DO-loop range in trying to optimize the load and store operations. The programmer can explicitly assist the compiler in the optimization process by using a particular programming style. This intervention involves minimizing, inside the DO-loop range, the work vectors used to store intermediate results and replacing them with fictitious scalar variables. This method forces the compiler to hold in the vector registers (when a sufficient number of them are available) the intermediate results of the computation. The programmer is assisted in the vector-register optimization technique by the listing of the object module in pseudo-assembler language that can be generated with the LIST compiler option. In general, successful optimization is enhanced by minimizing load and store operations.

The Vector Facility provides two compound instructions, *multiply-add* and *multiply-accumulate*, performing two operations in a single machine cycle. We have tried to increase the number of the compound instructions automatically generated by the compiler. Once the statements with compound instructions had been identified, we have verified by means of the object listing whether compound instructions had also been generated. Otherwise, we have tried to reorder the statement operation, keeping in mind that the compiler follows arithmetic rules in performing operations and that three operands cannot reside simultaneously in the vector registers.

Table 3 *Performance results*

Test Case (s)	Original-Version Scalar Run	Enhanced-Version Scalar Run	Enhanced-Version Vector Run
S1	1234	971	449
S2	6273	5303	1774

We present some performance results showing the improvement the migration work has yielded. In Tables 3 and 4 we report the performance results of two test cases in terms of CPU time for the first table and in terms of speedup for the second table. The two selected test cases were chosen to represent the lower and the upper bound in performance results,

Table 4 Performance results with speedup

Speedup-Enhanced Version	Speedup vs Original-Version Scalar Run	Speedup vs Enhanced-Version Scalar Run
S1	2.7	2.2
S2	3.5	3.0

Table 5 CPU percentages for scalar and vector modes

Code Section	CPU Percentage (Scalar Mode)	CPU Percentage (Vector Mode)
1 Element force evaluation	70–82	52–83
2 Geometry update	7–9	4–6
3 Explicit time integration	3–4	4–6
4 Contact algorithm	17–3	36–2
1 + 2 + 3 +4	97–98	96–97

Table 6 Processing time percentages and parallel percentages

Code Section	CPU Percentage (Vector Mode)	Parallel Percentage
1 Element force evaluation	82	74
2 Geometry update	5	5
3 Explicit time integration	6	0
4 Contact algorithm	3	0
1 + 2 + 3 + 4	96	79

due to the different incidence of vector code exploitation. In Table 3 the second column shows the CPU time needed by the original ESI scalar version run in scalar mode, the third shows the CPU time of the final enhanced-version result of the migration work, compiled and run in scalar mode, and the fourth shows the time of the same enhanced version run in vector mode.

In Table 4, the two righthand columns show the speedups of the final enhanced version run in vector mode compared to the original version and to the enhanced one, both run in scalar mode.

Parallel implementation

Once the vectorization was completed, we faced the problem of parallelizing the program by dividing up the computation among the processors in a multiprocessor configuration. The main objectives in developing a parallel version of the PAM-CRASH program were improvement of turnaround time in a shared environment and reduction of elapsed time in a dedicated environment.

The software selected to support the development of the PAM-CRASH parallel version was the Multitasking Facility (MTF), a standard feature of the IBM VS FORTRAN compiler. MTF uses the MVS macros to execute selected routines in parallel and to synchronize their execution. The programmer can easily introduce into the program suitable calls to the simple MTF primitives to specify the sections of the code to be executed in parallel and to synchronize their execution. The overhead associated with execution scheduling of a parallel subroutine is 75 μs.[25]

The percentage incidence, with respect to total processing time, of code sections which can take advantage of vector enabling and consequently are eligible to be parallelized drops considerably in analyses involving a reduced use of the contact algorithm. On the other hand, that percentage has remained nearly constant in analyses with a large utilization of serial code. In Table 5, the second column shows the processing time percentages associated with code sections executed in scalar mode, and the third one shows those associated with the same sections executed in vector mode. The two values of each column define the range of variation depending on the type of analysis and, consequently, on vector processing exploitation.

A parallel feasibility analysis was performed in order to define the conditions which allow parallelism to be efficiently exploited and to estimate costs, times, and resources. We selected a reference test case which needed sufficient computer resources to require utilization of parallel processing. The crash analysis of a complete vehicle—about 10 CPU hours with the vector version—was chosen. The criteria followed in identifying the code sections to be parallelized consisted in selecting the sections benefitted by the vector processing. The vector processing time of those sections had to exceed a threshold above which the overhead due to dispatching and synchronization could be considered negligible. The requirement was met for those code sections performing the *shell-element force evaluation* and the *geometry update.* Table 6 shows the processing time percentage of each single section and the parallel percentage we estimated for the reference test case, the crash analysis of a complete vehicle.

We estimated the parallel percentage, defined as the percentage of the vector execution time relative to those code sections which benefit from parallelization, at about 79 percent. A relation based on Amdahl's law was used to estimate the improvement in performance to be expected from parallel processing. That relation takes into account the degradation in performance due both to the task load imbalance and to the parallelization overhead, by means of specific normalized factors, respectively o_p and lb_p.

$$S = \frac{1}{(1 - f_p) + (f_p/n_p) * (1 + o_p + lb_p)}, \quad (2)$$

where S is the overall parallel speedup factor, f_p is the percentage of parallelization, n_p is the number of processors, o_p is the overhead factor associated with task scheduling and synchronization, and lb_p is the task load imbalance factor. The overhead factor o_p takes into account the overhead associated with task dispatching and synchronization; it is normalized with respect to the average execution time of each task,

$$o_p = \frac{t_0}{(f_p/n_p)}, \quad (3)$$

where t_0 is the time needed to perform the task scheduling and synchronization. The task load imbalance factor lb_p takes into account the possibility that the workload is not equally distributed among the parallel tasks. This condition is verified when more computational paths can be followed inside the parallel code.

The parameter lb_p is defined as follows:

$$lb_p = \frac{t_{max}}{f_p/n_p} - 1, \quad (4)$$

where t_{max} is the maximum execution time among the parallel tasks.

Assuming that the task load imbalance factor lb_p is equal to 0 (i.e., the workload is equally distributed among the parallel tasks) and imposing an overhead o_p not exceeding 5 percent, we have estimated the performance improvement to be 1.5 for 3090-200E systems and 2.4 for 3090-400E systems.

The algorithm of the shell-element force evaluation section is suitable for parallel processing, except for the section assembling the element forces into the global force vector, whose algorithm is serial. The solution selected consisted in assigning to each processor the force computation relative to 128 elements. The synchronization was scheduled before the force assembly was performed. The number of dispatching operations was equal to the number of 128 element groups, and the number of synchronizations was equal to the number of dispatching operations divided by the number of available processors.

From the point of view of the coding structure, an interface routine has been created to reorganize the data communication among the main task and the parallel ones. This operation was previously performed using the storage areas defined by the COMMON statement, whereas it is now carried out by means of the subroutine arguments. The element force assembly section has been moved into a new routine whose execution is scheduled after the synchronization. Figure 5 shows the implementation structure of the shell-element algorithm. This parallel migration phase was quite laborious because we were forced to apply heavy changes in the code structure and considerable effort went into preserving code readability.

The task load imbalance factor lb_p was discovered during the program test to be greater than 0. A load imbalance among the tasks arose because the shell element is a nonlinear element. Different computational paths can be followed, depending on the deformation and stress state of the structure, which generate a task load imbalance that does not remain constant but differs step by step. This factor negatively affected the performance of the parallel version. Performance measurements have further shown that the overhead factor does not exceed a value of 2 percent.

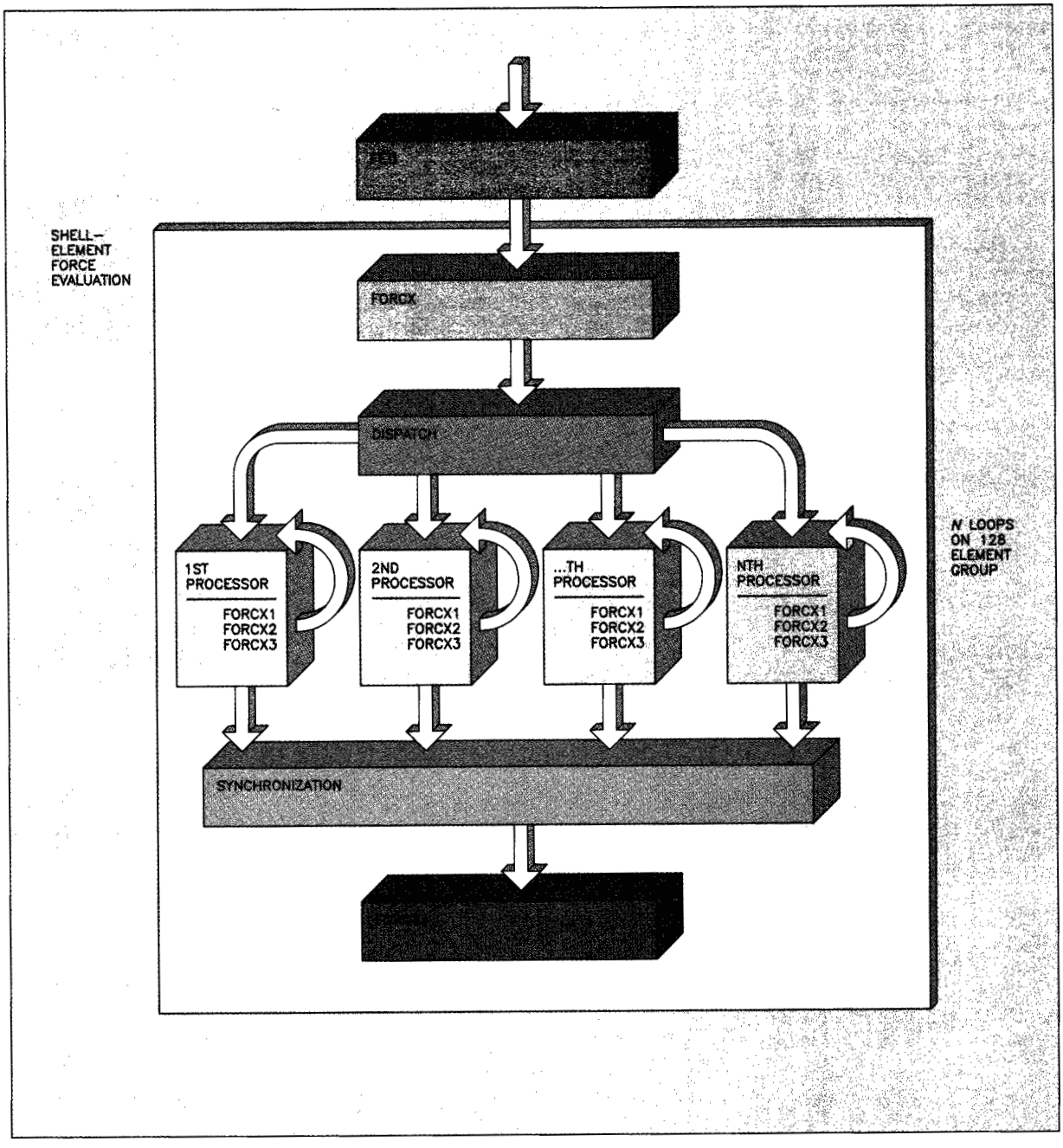
The *geometry update* section was parallelized by means of the MTF capabilities. Each parallel task performs the update in the finite-element model of a defined region whose dimension depends on the number of available processors.

Benchmarks were performed on the test case, which involved a complete vehicle model, and in other selected cases suitable for parallelization. The speedup figures, reported in Table 7, provide an average estimation of the parallel PAM-CRASH version performance.

PAM-CRASH graphic processing

A finite-element code can be successfully utilized only if it is provided with graphic programs aiding

Figure 5 Parallel implementation of the shell algorithm



the user in the pre- and post-processing phases. PAM-CRASH code is interfaced with the PRE-3D[26] preprocessor and the DAISY[27] postprocessor to form an integrated environment for crash analysis. Both codes were developed and are maintained by ESI; they provide powerful graphic capabilities to generate finite-element models and visualize finite-element analysis results. Use of the IBM GRAPHIGS Library[28,29] allows PRE-3D and DAISY to run on the IBM 5080 Graphics System.[30] They supply improved interac-

tivity by means of the 5080 local devices and performance improvement by exploiting the 5080 local processing capabilities:

- *5080 local devices*, e.g., graphics tablet and dials, allow the user to achieve a real-time interaction in performing image manipulation such as translation, rotation, clipping, and zooming. These transformations are performed by taking advantage of the 5080 local processing capabilities, with significant offloading of the host processor.
- *GRAPHIGS 3D functions*[31] allow the user to handle graphics data in a real 3D environment. In most of the engineering graphic codes, the visualization process is obtained by projecting the object to be displayed on an assigned plane that depends on the observer's position, and then applying 2D primitives to the projected object. GRAPHIGS primitives reduce the complexity and the dimensions of the application program by performing the projection process, which was previously a programmer's task, directly.

PRE-3D is a graphics preprocessor for the interactive generation of three-dimensional meshes; with the application of simple changes, it can easily be interfaced with other finite-element codes. The code was specifically designed either for use in generating finite-element models, or in adjusting and refining meshes, originally developed for static analyses, to make them suitable for crash analysis. PRE-3D provides the following capabilities to shorten and simplify the generation of a model:

- *Mesh generation:* Generation of lines and splines, generation of surfaces and volumes, intersections of volumes and surfaces
- *Mesh modification:* Interactive modification and refinement of meshes, interactive modification of element, material, and sliding surface characteristics
- *Mesh visualization:* Total and partial visualization of the model, with the capability of translating, rotating, and zooming the image

PAM-CRASH analysis results can be displayed by DAISY, providing a user-friendly interactive graphics environment. The capabilities provided by the code comprise the following:

- *Mesh visualization:* Total or partial perspective view of the model at each step of the analysis, with removal of either hidden lines or hidden surfaces and shaded-image display. Selective viewing of different materials or different elements with image zooming, scaling, rotating, and clipping features is also available.
- *Analysis result interpretation:* A powerful tool aiding the engineer in the model-verification phase. Either contour lines or color-filling patterns associated with stresses, strains, and kinematic entities are available at each step of the analysis, both for the complete model and for part of it. Effective stresses, according to different yielding criteria, can also be displayed. These values can also be plotted in a graph shape selecting the nodes of interest.

Table 7 Average estimated parallel PAM-CRASH performance

3090 System	Parallel Speedup
300E	2.0
400E	2.5
600E	3.0

Examples of the DAISY output are shown in Figures 6–11. They display different steps of the crash-simulation analysis of a moving vehicle striking a rigid wall.

Conclusions

Our work has resulted in the development of a new IBM version of PAM-CRASH which utilizes the vector and parallel capabilities of the 3090/VF system, together with new IBM versions of the PRE-3D and DAISY programs.

The new version of PAM-CRASH presents a performance improvement, compared to the old scalar performance, of 300 percent on a uniprocessor 3090E/VF, of 600 percent on a 3090-300E/VF, and of 900 percent on a 3090-600E/VF system.

The quality of the results, not only in terms of PAM-CRASH performance improvement, but also with respect to increased ease of use of the graphics package (due to stronger interactivity), has far exceeded initial expectations.

In October 1987, the new IBM versions of the programs were made available by ESI, which is now capable of handling the code implementation of future developments.

PAM-CRASH has been installed and runs successfully on the 3090/VF systems of a Japanese automotive manufacturer.

Figure 6 The complete model of the Citroen BX car displayed with shading effects

Figure 7 The image of a car model displayed with hidden line removal technique

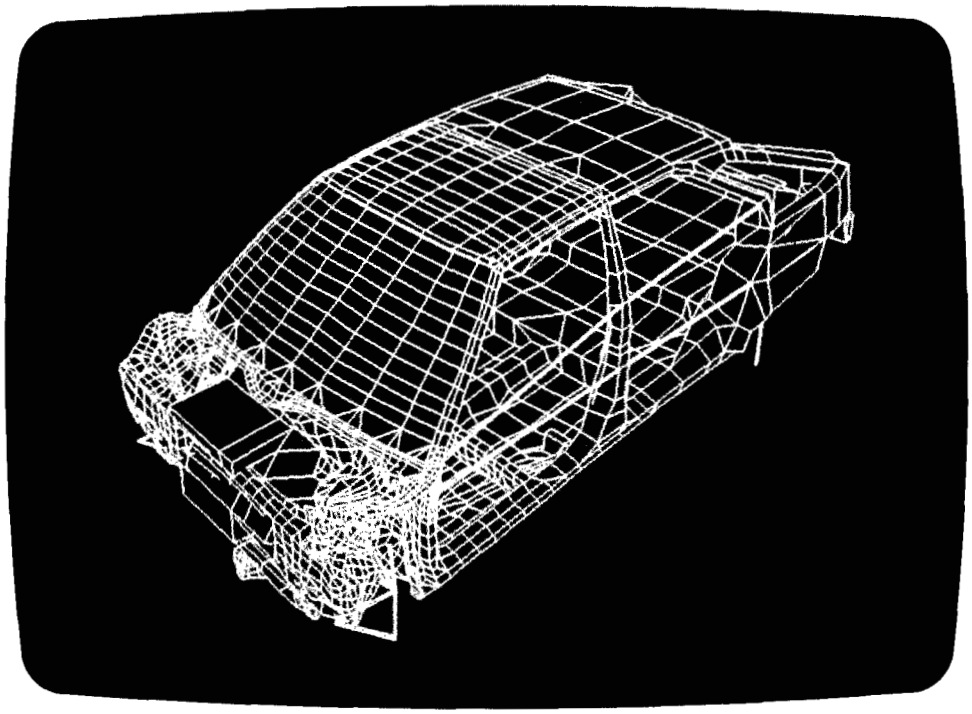


Figure 8 A wire frame model displayed during a session of an interactive 3-D manipulation

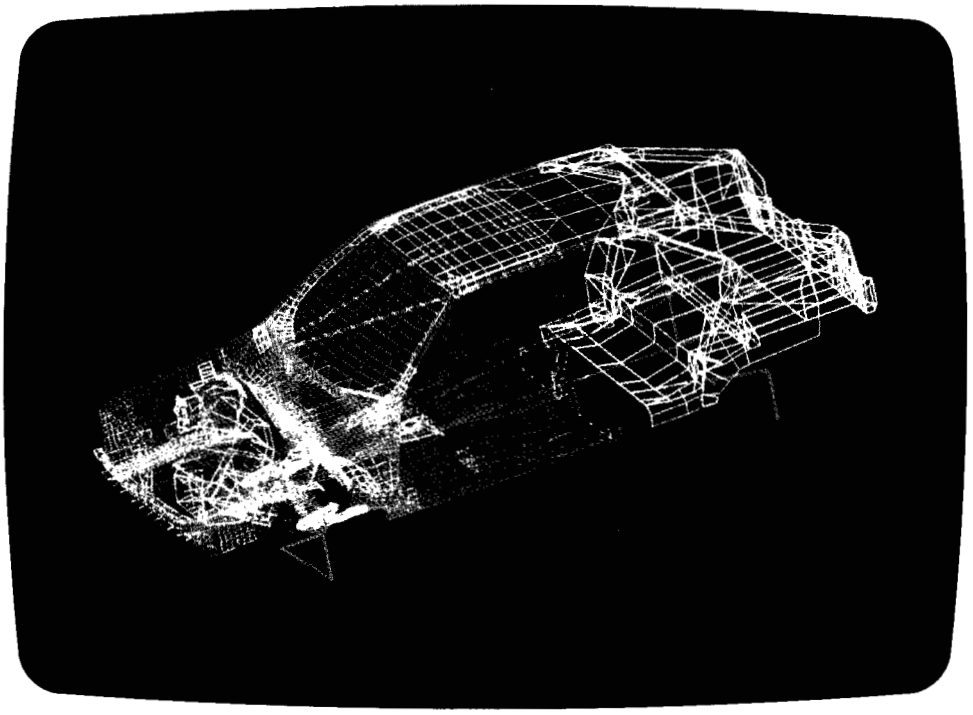


Figure 9 A velocity component colored field of the frontal vehicle section

Figure 10 Display of a velocity component of the vehicle compartment with an exploded view as a colored field

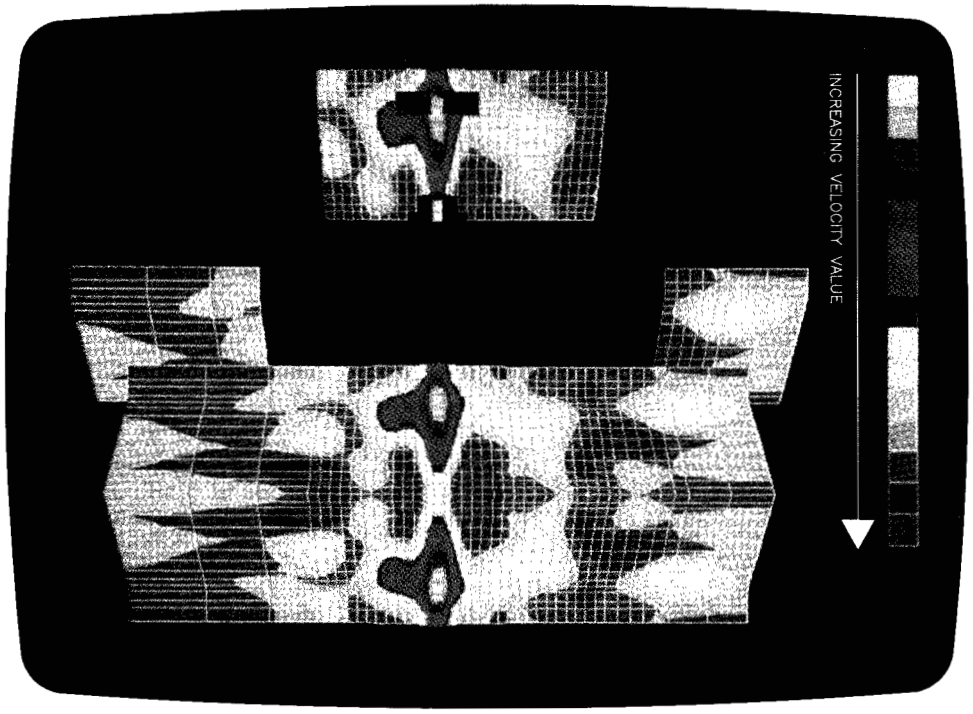
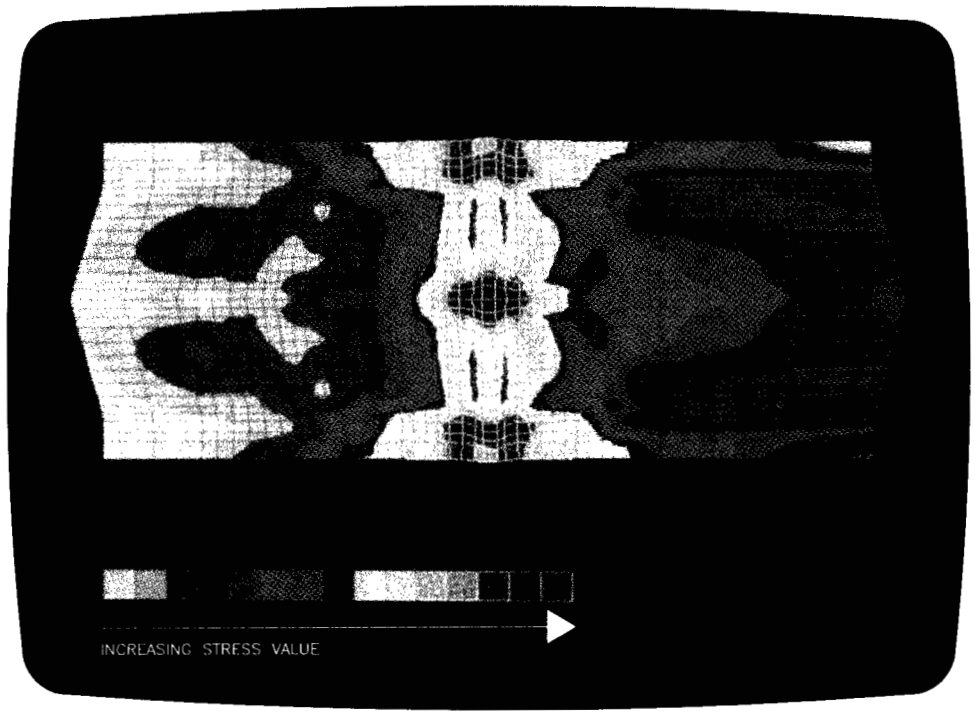


Figure 11 Display of a stress component of a vehicle as a colored field



Acknowledgments

We are grateful to Shunichi Katoh and Susumu Suda (IBM Japan) for their active contribution in developing, respectively, the vector and the parallel versions of PAM-CRASH. We are also grateful to Leo Carlesimo (IBM/ECSEC) and Cecile Pauquet (ESI) for their valuable cooperation in the migration of the graphics package, and to Renzo Di Antonio (IBM/ECSEC) for suggestions on the evaluation of system performance. Crash analyses on the Citroen BX automobile were carried out by the ESI team on the IBM 3090-600E Vector Facility of ECSEC. Finally, the authors are indebted to Silvia Frangipane (IBM/ECSEC) for her critical reading of the manuscript.

Cited references

1. *PAM-CRASH User Manual*, Engineering Systems International, Paris (1987).
2. J. F. Chedmail, P. Du Bois, A. K. Pickett, E. Haug, B. Dagba, and G. Winkelmuller, "Numerical techniques, experimental validation and industrial applications of structural impact and crashworthiness analysis with supercomputers for the automotive industries," *Proceedings of the International Conference on Supercomputer Applications in the Automotive Industry*, Zurich (1986).
3. T. J. R. Hughes, K. S. Pister, and R. L. Taylor, "Implicit-explicit finite elements in nonlinear transient analysis," *Computer Methods in Applied Mechanics and Engineering* **17/18,** 159–182 (1979).
4. T. Belytschko and J. I. Lin, "Explicit algorithms for the nonlinear dynamics of shells," *Computer Methods in Applied Mechanics and Engineering* **42,** 225–251 (1984).
5. T. Belytschko and C. S. Tsay, "A stabilization procedure for the quadrilateral plate element with one-point quadrature," *International Journal for Numerical Methods in Engineering* **19,** 405–419 (1983).
6. T. Belytschko, J. S. Ong, W. K. Liu, and J. M. Kennedy, "Hourglass control in linear and nonlinear problems," *Computer Methods in Applied Mechanics and Engineering* **43,** 251–276 (1984).
7. T. J. R. Hughes, R. L. Taylor, J. L. Sackman, A. Curnier, and W. Kanoknukulchai, "A finite element method for a class of contact-impact problems," *Computer Methods in Applied Mechanics and Engineering* **8,** 249–276 (1976).
8. G. L. Goudreau and J. O. Hallquist, "Recent developments in large-scale finite element Lagrangian hydrocode technology problems," *Computer Methods in Applied Mechanics and Engineering* **33,** 725–757 (1982).
9. J. O. Hallquist, G. L. Goudreau, and D. J. Benson, "Sliding surfaces with contact-impact in large-scale Lagrangian computations problems," *Computer Methods in Applied Mechanics and Engineering* **51,** 107–137 (1985).
10. *IBM System/370 Vector Operations*, SA22-7125, IBM Corporation; available through IBM branch offices.
11. W. Buchholz, "The IBM System/370 vector architecture," *IBM Systems Journal* **25,** 51–62 (1986).
12. *3090 Processor Complex: Functional Characteristics*, SA22-7121, IBM Corporation; available through IBM branch offices.
13. R. S. Clark and T. Wilson, "Vector system performance of the IBM 3090," *IBM Systems Journal* **25,** 63–82 (1986).
14. R. G. Scarborough and H. G. Kolsky, "A vectorizing FORTRAN compiler," *IBM Journal of Research and Development* **30,** 163–171 (1986).
15. *VS FORTRAN V2 Language and Library Reference*, SC26-4221, IBM Corporation; available through IBM branch offices.
16. *VS FORTRAN V2 Programming Guide*, SC26-4222, IBM Corporation; available through IBM branch offices.
17. *VS FORTRAN Execution Analyzer*, SC23-0335, IBM Corporation; available through IBM branch offices.
18. *Vectorization and Vector Migration Techniques*, SR20-4966, IBM Corporation; available through IBM branch offices.
19. E. Hinton and D. R. J. Owen, *Finite Elements in Plasticity*, Pineridge Press, Swansea, U.K. (1982).
20. K. J. Bathe, *Finite Element Procedures in Engineering Analysis*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1982).
21. *Finite Element Handbook*, H. Kardestuncer and D. H. Norrie, Editors, McGraw-Hill Book Co., Inc., New York (1987).
22. G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *AFIPS Conference Proceedings* **30,** 483–485 (1987).
23. *Designing and Writing FORTRAN Programs for Vector and Parallel Programs*, SC23-0337, IBM Corporation; available through IBM branch offices.
24. G. Radicati and M. Vitaletti, *Sparse Matrix-Vector Product and Storage Representation on the IBM 3090 Vector Facility*, IBM ECSEC Report G513-4098, European Center for Scientific and Engineering Computing, Rome (1986).
25. P. Carnevali, P. Sguazzero, and V. Zecca, "Microtasking on IBM multiprocessors," *IBM Journal of Research and Development* **30,** 574–582 (1986).
26. *PRE-3D User Manual*, Engineering Systems International (ESI) S.A. Rungis-France, Paris (1987).
27. *DAISY User Manual*, Engineering Systems International (ESI) S.A. Rungis-France, Paris (1987).
28. *Introducing GraPHIGS*, SC33-8100, IBM Corporation; available through IBM branch offices.
29. *Understanding GraPHIGS*, SC33-8102, IBM Corporation; available through IBM branch offices.
30. *IBM 5080 Graphics System: Principles of Operation*, GA23-2012, IBM Corporation; available through IBM branch offices.
31. *Programmer's Reference for GraPHIGS*, SC33-8104, IBM Corporation; available through IBM branch offices.

General reference

J. D. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Publishing Co., Reading, MA (1982).

P. Angeleri *IBM European Center for Scientific and Engineering Computing (ECSEC), via Giorgione 159, 00144 Rome, Italy.* Mr. Angeleri received his degree in civil engineering at the University of Rome in 1983. From 1983 to 1985 he worked as consulting engineer in bridge and aseismic structure design. At the same time, he was working in conjunction with the Department of Structural and Geotechnical Engineering of Rome University on the development of finite-element models for nonlinear dynamic analysis. In 1985 Mr. Angeleri joined IBM at the NIC Center in Rome, where he has been working on finite-element method developments and vector/parallel migration.

D. Fabio Lozupone *IBM European Center for Scientific and Engineering Computing (ECSEC), via Giorgione 159, 00144 Rome, Italy.* Mr. Lozupone received his degree in civil engineering at the University of Rome in 1983. From 1983 to 1987 he worked as a consulting engineer for bridge and aseismic structure design. At the same time, he was working with the Department of Structural and Geotechnical Engineering of Rome University, primarily on earthquake engineering and soil-structure interaction problems. In 1987 Mr. Lozupone joined IBM at the NIC Center in Rome, where he has been working on finite-element method development and applications and on graphics representation of engineering code results.

F. Piccolo *IBM European Center for Scientific and Engineering Computing (ECSEC), via Giorgione 159, 00144 Rome, Italy.* Mr. Piccolo received his degree in physics at the University of Rome in 1986. From 1986 to 1987, he worked as an aerospace engineer on the ESA-Remote Sensing Satellite-1. In 1987 Mr. Piccolo joined IBM at the NIC Center in Rome, where he has been working on finite-element method applications and on graphics representation of engineering code results.

Jan Clinckemaillie *Engineering Systems International (ESI), 20, Rue Saarinen, 94578 Rungis, France.* Mr. Clinckemaillie obtained his degree in civil engineering at the University of Ghent, Belgium, in 1979. Between 1979 and 1981 he studied at the University of California, Berkeley, as a Fulbright scholar and received the degrees of Master of Science and Master of Engineering. He joined Engineering Systems International in 1982, and has been involved in the development and application of finite-element codes in various fields such as hydraulic fracturing, lubrication, inflatable structures, etc. At present Mr. Clinckemaillie is program manager of ESI's hydrodynamics code EFHYD and its crash-analysis code, PAM-CRASH.

Reprint Order No. G321-5342.