Improving Computational Efficiency of VAST

Lei Jiang and Tom Macadam

Martec Limited

Prepared By:
Martec Limited
400-1800 Brunswick Street
Halifax, Nova Scotia
B3J 3J8 Canada

Contract Project Manager: Lei Jiang, 902-425-5101 Ext 228
Contract Number: W7707-125422/001/HAL CU09
CSA: Malcolm J Smith, 902-426-3100 Ext 383

Contract Report
DRDC-RDDC-2014-C34
MARTEC TR-13-42
September 2013

Principal Author

Lei Jiang
Senior Research Engineer

Abstract

This report describes development and implementation of an in-core database in the VAST finite element program. The database was developed based on the C++ Standard Template Library (STL) generic data structures and a set of Application Programming Interface (API) functions were provided to allow access from Fortran. The arguments of the API functions were designed to be of a generic format, to minimize the number of functions and permit easy future expansion. Implementation of the database required significant restructuring of VAST code. A pre-processor module, named PREPR1, was developed to import all the input data and store them into the database. During the VAST executions, the database was not only used as the source of the finite element model data, but also used as the temporary storage of many of the intermediate results. This treatment of data flow eliminated a large amount of I/O operations in the original VAST program and resulted in very significant savings on the computation time. The resulting version of VAST has been extensively verified and benchmarked using test problems of different sizes and the benchmark results have indicated that by combining the new database and the new sparse solver, the overall speed of VAST was increased by a factor of five for engineering problems. The current API version of VAST has some limitations, such as limited element types and analysis capabilities as well as a size limit on the finite element model. Further tasks are recommended for removing these limitations.

This page intentionally left blank.

Executive summary

Improving Computational Efficiency of VAST:

Introduction: VAST is a general-purpose nonlinear finite element solver program developed and maintained by Martec over the past four decades under the sponsorship of DRDC Atlantic and has been adopted as the built-in default finite element solver in a number of structural analysis packages, such as Trident, SubSAS and CRS STRUC. In addition, the VAST has been used to support development of customized solutions, such as systems for analyzing the mast structure on naval vessels (MAST), propeller structural loading (PVAST) and ultimate strength analysis (UltSAS). VAST was originally developed in the early 1970s, and designed to solve large problems on small (low RAM) single CPU machines. In order to achieve this goal, VAST was purposely structured in modules which communicated through disk files, resulting in a very large volume of I/O operations which significantly degraded the computational efficiency. Two approaches can be taken to improve the computational efficiency. One is to implement the modern parallel techniques, and the other is to optimize the data flow by minimizing the I/O operations. The second approach was addressed in the present work.

Results: An in-core database was developed and implemented in VAST. This database was developed based on the C++ Standard Template Library (STL) generic data structures and a set of Application Programming Interface (API) functions were provided to permit access from a Fortran program. The arguments of the API functions were designed to be of a generic format, to minimize the number of functions and permit easy future expansion. The VAST program was significantly restructured to accommodate the new database, including development of a pre-processor module, named PREPR1. During the VAST executions, the database was not only used as the source of the finite element model data, but also used as the temporary storage of many of the intermediate results. The resulting version of VAST was extensively tested and benchmarked using problems of different sizes and the benchmark results have indicated that by combining the new database with the newly improved sparse solver, the overall speed of VAST was increased by a factor of five in practical engineering analyses.

Significance: The benchmark results clearly demonstrated the potential for generating a highly efficient version of VAST in the future by adopting a properly designed database and using the improved sparse solver. These findings provide directions for future investigations on efficiency improvement. In particular, the current API version is ideal for use as a test bed for these explorations.

Future plans: Although it has been demonstrated that the present API version of VAST was very efficient, it only provides limited capabilities and is only capable of solving relatively small problems. A number of tasks for future investigation have been recommended. These included expanding the database to include out-of-core processing; parallelizing modules such as element formulation, matrix assembly and stress calculations, and improving the robustness of the new sparse solver. Once these problems are resolved successfully, all the capabilities provided in the full version of VAST need to be imported to the highly efficient API version.

This page intentionally left blank.

Table of contents

List of figures

List of tables

This page intentionally left blank.

1 Introduction

VAST is a general-purpose nonlinear finite element solver program developed and maintained by Martec over the past four decades under the sponsorship of DRDC Atlantic [1]. The current version of VAST provides a large selection of element formulations, material models and analysis options. These computational capabilities have been extensively verified and validated using analytical and experimental results and have been utilized successfully in numerous practical engineering analyses, such as plastic collapse of submarine pressure hulls and crack initiation and propagation in various naval structures. At the present time, VAST has been adopted as the built-in default finite element solver in a number of structural analysis packages, such as Trident, SubSAS and CRS STRUC. In addition, the VAST program has been used to support development of customized solutions not possible with commercial codes. Examples include development of systems for analyzing the mast structure on naval vessels (MAST), propeller structural loading (PVAST) and ultimate strength analysis (UltSAS).

VAST was originally developed in the early 1970s, and designed to solve large problems on small (low RAM) single CPU machines. In order to achieve this goal, VAST was purposely structured in modules. Each module performed a particular step in finite element analyses, such as element matrix formulation, assembly and decomposition of the global stiffness matrix, generation of load vectors, solution of displacement vectors and evaluation of element stresses. In order to permit restart between the modules, large volume of intermediate results from the finite element calculations were stored on disk files, resulting in huge amounts of I/O operations which significantly degraded the computational efficiency. With the advent of multi-core symmetric processor (SMP) machines and the significant increases in RAM, especially on the 64-bit architecture, it is now possible to modernize VAST to utilize this large increase in computing power to solve problems that are increasingly demanding.

Two approaches can be taken to improve the computational efficiency of a finite element code, like VAST. One is to implement the modern parallel techniques, such as the OpenMP and MPI, to speed up the generation of element matrices, solution of linear algebraic equations and evaluation of stresses. These requirements have been partially addressed through parallelization of the direct sparse matrix solver [2,3].

The other modifications required for VAST include optimization of its data flow by minimizing the I/O operations. This requires development and implementation of a highly efficient database for VAST by taking advantage of the massive RAM space potentially available on computers running a 64-bit operation system. This data structure will hold the input information, the intermediate results and the solutions mostly in memory during finite element computations, so that the disk I/O operations are only performed when they are absolutely necessary. The elimination of unnecessary disk I/O is expected to significantly improve the efficiency of VAST, especially for nonlinear analyses where extensive I/O operations are used in current version of VAST to facilitate the Newton-Raphson iterations. The present contract is the first attempt to optimize the data flows in VAST for improved efficiency.

Three objectives to be accomplished in the present contract are:

1. Develop an in-core data structure for VAST. This data structure will be highly efficient and fit naturally into the overall operations in the VAST program. In the meantime, it will be sufficiently general to support all element types, material models and analysis capabilities currently provided in VAST and will be expandable to permit future developments of new VAST capabilities.

2. Implement the data structure into VAST. The database is accessible from the Fortran code though a set of API functions. In order to maximize benefit from the new data structure, the VAST program will be restructured. The initial implementation will focus on the most commonly used element types in ship structure analyses, including 2-noded beam and 4-noded quadrilateral shell elements. Both linear and nonlinear static analysis capabilities will be provided.

3. Verify and benchmark the new version of VAST. Upon completion, the new version of VAST with the data structure will be extensively verified using the standard test cases from the VAST Autotester and larger finite element problems utilized in previous practical engineering analyses. The computational efficiency of the improved version will be benchmarked against that of the original version of VAST.

In this report, the development of the database and its implementation in VAST will be discussed in the next chapter. The results of verification and benchmark of the new version of VAST are presented in Chapter 3. The conclusions and recommendations for future work are outlined in Chapter 4.

2 Development and Implementation of a Database for VAST

2.1 Development of API functions

During this phase of the project, work was undertaken to restructure the VAST source code so that all reading and writing of model data was directed through a single collection of subroutines. This collection of subroutines was termed the VAST Data Application Programming Interface (referred to as the API in the remainder of this document). The purpose of restructuring VAST and introducing the API was to collect and isolate the details concerning storage of model data behind a static subroutine interface so that changes could be made to the storage mechanism without affecting the rest of the VAST code beyond the API layer (see Figure 1). Primarily, these changes to VAST separated data flow involved in finite element analyses from the actual computations. It not only eliminated a large number of I/O operations, but also permitted further optimization of algorithmic operations. Secondly, the work improved the modularity of VAST so that different parts could undergo major changes independently. Thirdly, it focused all data access to the API, thereby making it easier to add logging and instrumentation in a central location to track data access patterns and performance.

The first part of this work comprised defining the API functions and their associated parameters. A complete list of the current API subroutines is given in Table 1. In order to keep the number of subroutines in the API low, it was decided that the subroutines would not be parameterized on specific types of model entities, but rather on overall categories of entities. For instance, rather than a variety of subroutines to set property data for each type of finite elements, there was one generic subroutine to set property data for all element types. To accommodate this strategy, the subroutines accepted most arguments as arrays of generic integer or floating point data. It was up to VAST to load/unload the values from these arrays and to establish/re-establish their intended meaning. Not only did this result in fewer generic subroutines comprising the API, but it also promoted consistency and reuse of generic data patterns for storing the data below the API layer.

Once the API layer subroutines were fixed, a means to store the data being passed through the subroutines had to be put in place. For this phase of the project, an entirely in-core storage system was devised. A survey of reusable native Fortran data structures was undertaken, but yielded a poor selection of options. As such, it was decided to use the C++ Standard Template Library (STL) [4] generic data structures through an inter-language wrapper layer. Since most data could be mapped to key-value pairs (integer keys, array values), the STL hash table container, unordered_map<>, was used. Though mixing the languages in this way was recognized as not ideal, it was felt that the flexibility and robustness benefits of the STL outweighed any small overhead incurred calling through the wrapper layer.

Once the data being passed through the API was being stored and successfully retrieved, the prototype API was effectively complete. All areas in VAST that had previously written and retrieved data directly to/from files were updated to make use of the appropriate API subroutines. The correctness of the API functions was confirmed by test problems to be discussed later in this report. Because the storage system implemented in this Phase of the work was entirely in-core, it

exhibited comparatively high performance, but it was recognized this came at the cost of data volatility and model size limitations.

The next phase of the API development proposes to replace the in-core data storage with an out-of-core mechanism to allow for even larger model sizes and to provide for data persistence. Some initial research into methods used to efficiently store and retrieve large-scale numerical data from secondary storage (i.e. disk) has pointed towards the field of Scientific Data Formats. In short, these data formats are purpose-built, mainly by scientists and researchers, to efficiently work with vast volumes of dense numerical data. Various precedents have been set for using such data formats for storage of finite element analysis data [5] [6]. Early indications pointed towards the Hierarchical Data Format version 5 (HDF5) [7] as the most suitable format for consideration owing to its generality, maturity and widespread adoption. Some initial work has been done in parallel to this project to explore the format and try to identify an optimal storage arrangement to support the API version of VAST [8]. It is proposed to continue this work and further explore possible HDF5-based formats already tailored for FE data.

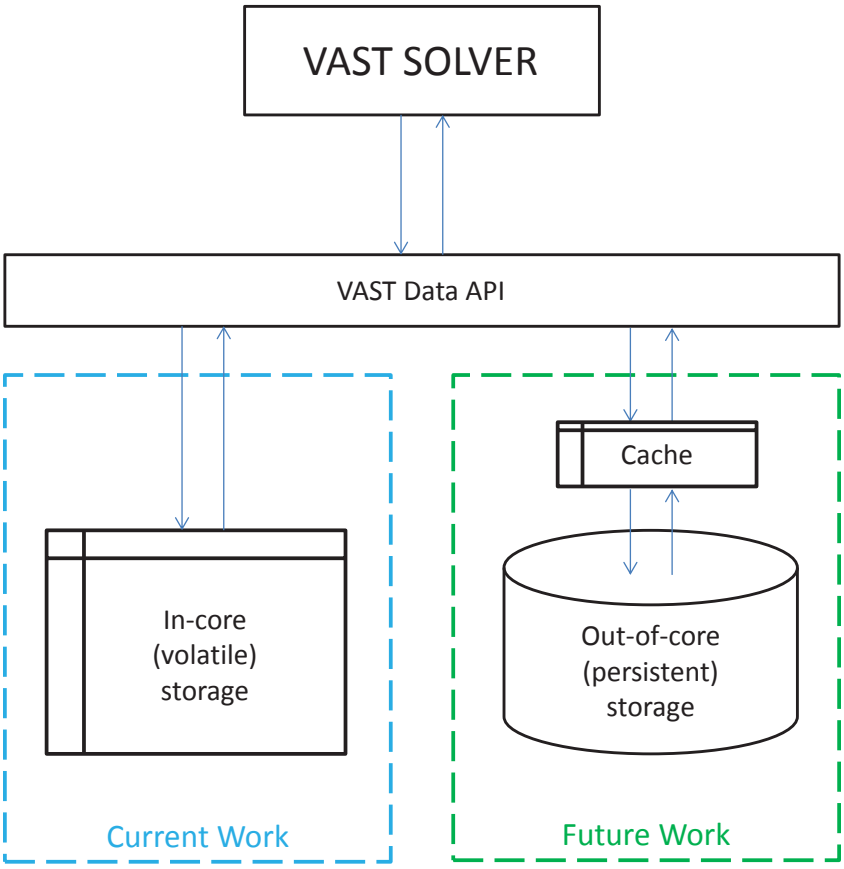

Figure 1: Model of data exchange between VAST and the database

Table 1: List of API functions for VAST database

```
api_initialize(int& ierr);
api_terminate(int& ierr);
api_dumptocout(int& ierr);

get_node( int& id_m, int& ni, int*& iarray, int& nr, double*& rarray ) { }
get_node_e( int& eid_m, int& ni, int*& iarray, int& nr, double*& rarray, int& ierr ) { }
set_node( int& id_m, int& eid_m, int& ni, int*  iarray, int& nr, double*  rarray, int& ierr ) { }

get_mat ( int& id_m, int& ni, int*& iarray, int& nr, double*& rarray ) { }
set_mat ( int& id_m, int& ni, int*  iarray, int& nr, double*  rarray ) { }

get_prop( int& id_p, int& ni, int*& iarray, int& nr, double*& rarray ) { }
set_prop( int& id_p, int& ni, int*  iarray, int& nr, double*  rarray ) { }

get_elem_main( int& id_e, int& ni, int*& iarray, int& nr, double*& rarray ) { }
get_elem_main_e( int& eid_e, int& ni, int*& iarray, int& nr, double*& rarray, int& ierr ) { }
set_elem_main( int& id_e, int& eid_e, int& ni, int*  iarray, int& nr, double*  rarray, int& ierr ) { }

get_elem_aux ( int& id_e, int& ni, int*& iarray, int& nr, double*& darray ) { }
set_elem_aux ( int& id_e, int& ni, int*  iarray, int& nr, double*  darray ) { }

get_elem_strs( int& id_e, int& ni, int*& iarray, int& nr, double*& darray ) { }
set_elem_strs( int& id_e, int& ni, int*  iarray, int& nr, double*  darray ) { }

get_elem_strs_i( int& id_e, int& ni, int*& iarray, int& nr, double*& darray ) { }
set_elem_strs_i( int& id_e, int& ni, int*  iarray, int& nr, double*  darray ) { }

get_elem_temp( int& id_e, int& ni, int*& iarray, int& nr, double*& darray ) { }
set_elem_temp( int& id_e, int& ni, int*  iarray, int& nr, double*  darray ) { }

get_elem_intf( int& id_e, int& ni, int*& iarray, int& nr, double*& darray ) { }
set_elem_intf( int& id_e, int& ni, int*  iarray, int& nr, double*  darray ) { }

get_cons_spc ( int& id_m, int& ni, int*& iarray, int& nr, double*& rarray ) { }
set_cons_spc ( int& id_m, int& ni, int*  iarray, int& nr, double*  rarray ) { }

get_cons_mpc ( int& id_m, int& ni, int*& iarray, int& nr, double*& rarray ) { }
set_cons_mpc ( int& id_m, int& ni, int*  iarray, int& nr, double*  rarray ) { }

get_cons_rlnk( int& id_m, int& ni, int*& iarray, int& nr, double*& rarray ) { }
set_cons_rlnk( int& id_m, int& ni, int*  iarray, int& nr, double*  rarray ) { }

get_cons_rbe3( int& id_m, int& ni, int*& iarray, int& nr, double*& rarray ) { }
set_cons_rbe3( int& id_m, int& ni, int*  iarray, int& nr, double*  rarray ) { }

get_mass_lump( int& id_m, int& ni, int*& iarray, int& nr, double*& rarray ) { }
set_mass_lump( int& id_m, int& ni, int*  iarray, int& nr, double*  rarray ) { }

get_strs( int& id_m, int& ni, int*& iarray, int& nr, double*& rarray ) { }
set_strs( int& id_m, int& ni, int*  iarray, int& nr, double*  rarray ) { }
```

Table 1 (Cont'd): List of API functions for VAST database

```
get_load_elem( int& id_m, int& ni, int*& iarray, int& nr, double*& rarray ) { }
get_load_elem_e_num( int& eid_m, int& num ) { }
get_load_elem_e( int& eid_m, int& idx, int& ni, int*& iarray, int& nr, double*& rarray, int& ierr ) { }
set_load_elem( int& id_m, int& eid_m, int& ni, int*  iarray, int& nr, double*  rarray ) { }

get_load_conc( int& id_m, int& ni, int*& iarray, int& nr, double*& rarray ) { }
get_load_conc_e_num( int& eid_m, int& num ) { }
get_load_conc_e( int& eid_m, int& idx, int& ni, int*& iarray, int& nr, double*& rarray, int& ierr ) { }
set_load_conc( int& id_m, int& eid_m, int& ni, int*  iarray, int& nr, double*  rarray ) { }

get_load_prdp( int& id_m, int& ni, int*& iarray, int& nr, double*& rarray ) { }
get_load_prdp_e_num( int& eid_m, int& num ) { }
get_load_prdp_e( int& eid_m, int& idx, int& ni, int*& iarray, int& nr, double*& rarray, int& ierr ) { }
set_load_prdp( int& id_m, int& eid_m, int& ni, int*  iarray, int& nr, double*  rarray ) { }

get_load_glob( int& id_m, int& ni, int*& iarray, int& nr, double*& darray ) { }
set_load_glob( int& id_m, int& ni, int*  iarray, int& nr, double*  darray ) { }

get_load_thrm( int& id_m, int& ni, int*& iarray, int& nr, double*& darray ) { }
set_load_thrm( int& id_m, int& ni, int*  iarray, int& nr, double*  darray ) { }

get_load_intf( int& id_m, int& ni, int*& iarray, int& nr, double*& darray ) { }
set_load_intf( int& id_m, int& ni, int*  iarray, int& nr, double*  darray ) { }

get_load_timf( int& id_m, int& ni, int*& iarray, int& nr, double*& rarray ) { }
set_load_timf( int& id_m, int& ni, int*  iarray, int& nr, double*  rarray ) { }

get_disp( int& id_m, int& ni, int*& iarray, int& nr, double*& darray ) { }
set_disp( int& id_m, int& ni, int*  iarray, int& nr, double*  darray ) { }

get_disp_i( int& id_m, int& ni, int*& iarray, int& nr, double*& darray ) { }
set_disp_i( int& id_m, int& ni, int*  iarray, int& nr, double*  darray ) { }

get_mode( int& id_m, int& ni, int*& iarray, int& nr, double*& darray ) { }
set_mode( int& id_m, int& ni, int*  iarray, int& nr, double*  darray ) { }
```

2.2 Implementation of API functions

The flowchart in Figure 2 below shows the main structure of the version of VAST with the new database. In this figure, the blue lines and arrows indicate the sequence of execution of different computational modules in VAST, whereas the red lines and arrows indicate the data flow in and out of the memory-based database. For nonlinear analyses, the modules are executed repeatedly for each equilibrium iteration in each solution step, until the analysis is completed.

The implementation of the database required substantial restructuring of the VAST program. The very first step for implementing the new database was to develop a pre-processor module, named PREPR1, which read all the VAST input files and stored the information in the database. These VAST files included the finite element model geometry (GOM), the boundary conditions (SMD), the load definitions (LOD), the lumped masses (MAS) and the solution control data (USE) file. The API functions started with set_ were utilized to pass data to the database. All the solution control parameters were treated as global variables and kept in a central common block. Once the database was populated, the text input data files were no longer required and all further operations of VAST were performed based on the database.

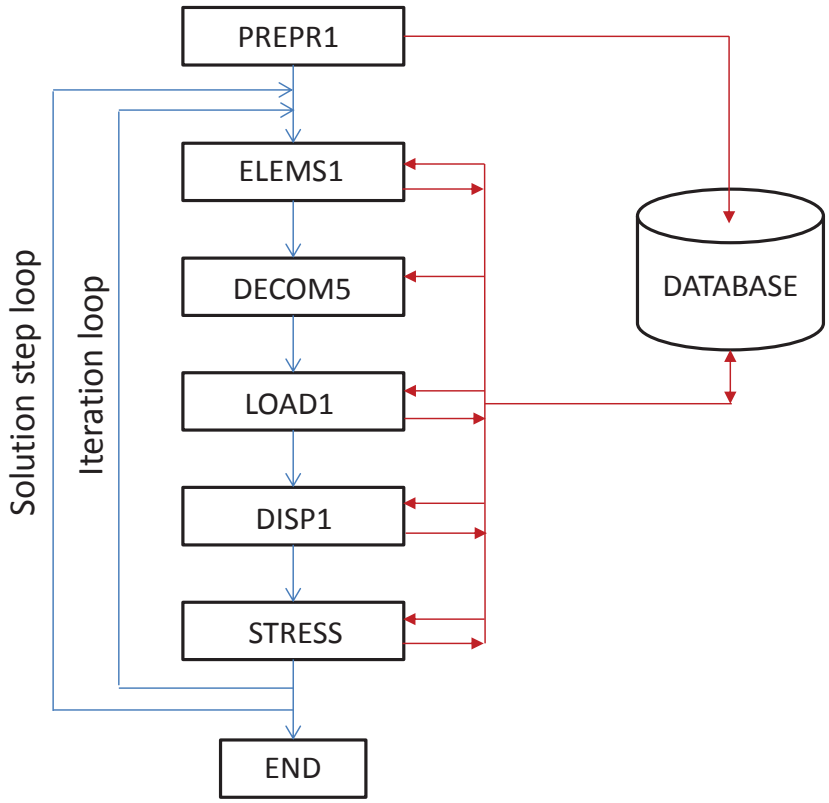

Figure 2: Main structure of VAST with the new database

In element module ELEMS1, element matrices were generated using the nodal coordinates, material properties and geometric properties extracted from the database. These were achieved by using API functions get_node, get_elem_main, get_mat and get_prop. The element matrices were then passed to the sparse solver through the Fortran/C++ interface as in the previous version of VAST. It should be noted that in the new database, nodes and elements can be accessed through either internal or external IDs. This arrangement provides a convenient means for VAST to perform analyses using models for other finite element programs, such as NASTRAN, where the external node and element IDs were normally used. For nonlinear analyses, geometric stiffness matrix and interface vector for each element were generated by using the most recently updated stresses and the other material variables extracted from the database via get_elem_strs. For the first iteration in each solution step, these stresses and material variables were stored back onto the database through set_elem_strs_i as they were required in subsequent iterations of the current load step. In addition, some intermediate results that were formulated during element generation and also required later in stress calculations were also saved on the database using set_elem_aux.

In assembly and decomposition module DECOM5, the boundary conditions and other constraint equations, such as the multi-point constraints and rigid links were retrieved from the database. This was accomplished by calling API functions get_cons_spc, get_cons_mpc and cons_rlnk. These constraint equations resulted in additional stiffness terms for the global stiffness matrix and these terms were passed to the sparse solver as in the previous version of VAST. Matrix assembly and factorization were then performed by the sparse solver. There was no information to be written back onto the database from this module.

In the load module LOAD1, equivalent nodal force vectors were formulated using the concentrated forces, element pressures and prescribed displacements stored in the database. This load information was retrieved using functions get_load_conc, get_load_elem and get_load_prdp. Similar to the storage for nodes and elements in the database, the load entries were also associated with internal and external IDs. This permitted efficient search of the load data for a given set of load cases through the application of hash functions. Once the global nodal force vectors were created, they were stored back onto the database via set_load_glob.

The displacement module DISP1 computed nodal displacements using the equivalent global nodal force vectors extracted from the database using get_load_glob. These load vectors were passed to the sparse solver through the Fortran/C++ interface and the sparse solver returned the corresponding displacement vectors. These displacements were then stored in the database using set_disp for stress calculations. For nonlinear analyses, the most recent global internal force vector was formed by first extracting the element internal force vectors using get_elem_intf and then assembling them into a global vector. This global internal force vector was also stored in the database using set_load_intf. In order to make the original and modified arc-length methods operational, the incremental displacements in the current solution step and current iteration must be saved in the database. These were achieved by using API functions set_disp and set_disp_i, respectively. In the meantime, the displacement vectors were also permanently stored on disk file Prefx.V52.

In the STRESS module, both the displacement vectors and the element information, such as node coordinates, element connectivity, material and geometric properties, were extracted from the database. The element stress matrices were then formulated and the stresses were calculated. The

calculated stresses were stored in the database through set_elem_strs and on disk file Prefx.V53 as a user option. Some matrices which were generated as intermediate results in element matrix formulations were also imported from the database through get_elem_aux and used in stress calculations. For nonlinear analyses involving elastic-plastic material behaviour, the stresses and plastic variables at the beginning of the current step were required for properly integrating the constitutive relation. In this case, these initial stresses and initial values of the plastic variables were obtained from the database by calling get_elem_strs_i.

Due to the implementation of the database, almost all of the intermediate binary files required by VAST were eliminated. The benchmark results to be presented later in this report indicated that the elimination of the I/O operations resulted in dramatic improvement on the computational efficiency of VAST.

2.3 Limitations of the present API version of VAST

The present API version of VAST contained two element types, 2-noded general beam and 4-noded quad shell elements. These elements were selected because they were the most commonly used elements in submarine modelling and were supported by SubSAS. Once fully verified, this highly efficiency API version of VAST would be ready for being used to replace the existing version of VAST in SubSAS.

In addition to the limited element library, the present API version of VAST was also restricted to linear and nonlinear quasi-static analyses. However, it permitted the full line of nonlinear material types, such as the elastic-plastic material models with bi-linear and piecewise linear stress-strain curves. All the nonlinear solution algorithms, including the orthogonal trajectory and modified arc-length methods, were supported. Some useful features for nonlinear collapse analyses, such as automatic restart and automatic adjustment of solution step, were also maintained. However, other analysis capabilities supported by the full version of VAST would have to be implemented in the future.

Besides the implementation of the database, the recently improved sparse direct matrix solver based on the super-node technology [2] was also incorporated. The improved sparse solver could be invoked by a new solution control parameter, IPARL. When IPARL>1, the new sparse solver would be activated. If IPARL=1, the original sparse solver would be utilized.

Although the present API version of VAST only provides limited capabilities, it is ideally suitable for use as a test-bed to explore various approaches for further speeding up its execution. This is partially due to the simplicity of its current code structure. The subjects of exploration should include parallelization of element generation, matrix assembly, load calculation and stress evaluation. In addition, in order to solve larger problems, the database needs to be expanded to use combined in-core and out-of-core operations. These issues will be discussed in more detail in the final chapter of the report.

3 Verification and Benmarking of API Version of VAST

3.1 Small test cases from VAST Autotester

The first set of test cases contained three small test problems chosen from the VAST Autotester [9]. Among them, CS03B involved nonlinear collapse of a simply-supported shallow arch under a centre concentrated force, whereas CS05B and CS05H dealt with collapse of a hinged shallow spherical shell subjected to a load at the pole. All three test cases involved elastic-plastic material model, but CS05B used a bi-linear stress-strain relation and both CS03B and CS05H used piece-wise stress-strain curves. Both the displacement control and the orthogonal trajectory methods were employed to obtain the complete nonlinear solutions.

The run times summarized in Tables 2-4 suggested that the combination of the new database and the improved sparse solver resulted in a consistent 20% reduction of the total run time for these small problems. However, due to their small sizes, these problems were not suitable for benchmarking the speed, but ensuring the correctness of the nonlinear solutions. In the present work, the results from the API version of VAST were carefully compared with those generated by the full version of VAST. The results were found to be identical. The deformed configurations and load-displacement curves obtained for these test cases are presented in Figures 3-6.

3.2 Mid-sized test cases of stiffened panels

The second group of test cases involved stiffened panel structures subjected to various load cases. These panels were analyzed previously in a number of LR internal projects [10,11], so the sizes of the finite element models were consistent with those utilized in practical collapse analyses. In the present study, we first performed a series of linear elastic analyses of panel FB6 using models of different levels of refinement shown in Figures 7-9. The results presented in Tables 5-7 clearly indicate that use of the new database and the improved sparse solver result in more significant savings on larger models. For models of 4,000 to 50,000 nodes, 50% to 70% reduction of total run time was obtained which corresponded to speed-up by factors of 2 to 3. Once again, the results from all versions of VAST were identical.

Following the linear elastic analyses, the plastic collapse behaviour of panel FB6 under uniform compression in the transverse direction was computed. The final deformed configuration and the transverse load-shortening curve are presented in Figures 10 and 11, respectively. The results from all VAST analyses were identical. The time results in Table 8 indicated a 50% saving on total run time which was consistent with the figure observed in linear elastic analysis (in Table 5).

The second stiffener panel considered was the L10 panel which involved L-shaped stiffeners as shown in Figure 12. Nonlinear collapse analysis of this panel under axial load was performed using different versions of VAST and the predicted final collapse mode and the load-shortening curve are presented in Figures 13 and 14, respectively. Due to the extremely strong nonlinearity in this problem, the nonlinear algorithm failed to converge near the limit load, but the automatic restart capability was invoked to continuously reduce the solution step until convergence was achieved successfully. The results from all versions of VAST were identical and times taken by these runs are compared in Table 9, where a 50% reduction on total run time is also obtained.

The final test case of stiffener panel also involved panel L10, but this time, a pressure was first applied to the structure to generate initial deformations. While this pressure was maintained at a constant level, a uniform transverse compressive stress was applied. This transverse stress was then increased gradually until the panel collapsed. In order to accurately represent the loading sequence, the applied pressure load and transverse stress must be arranged into different load cases and were controlled by independent load parameters. At the beginning of the nonlinear run, the pressure was activated and the load parameter was increased to the desired value. The active load case was then switched to the transverse stress and the nonlinear analysis was restarted. The purpose of this particular test case was utilized to verify the capability of the API version of VAST for dealing with such a complicated loading history that involved multiple load cases. Once again, the results from all versions of VAST were found to be identical. The final collapse mode and the transverse load-displacement curve are presented in Figures 15 and 16. It should be noticed that the flat part at the beginning of the load-displacement was due to the deformations caused by the pressure which was applied prior to the application of the transverse stress. The times taken by each of the VAST runs are compared in Table 10, where a reduction of 55% was observed.

3.3 Larger test cases of submarine structures

In order to benchmark the performance of the various versions of VAST for large finite element problems, linear and nonlinear analyses of a submarine compartment was conducted. The model utilized in these analyses was generated by using SubSAS in a previous DRDC contract [12] and composed of 4-noded shell and 2-noded beam elements as shown in Figure 17. The times taken by the various versions of VAST for linear elastic and nonlinear plastic collapse analyses are summarized in Tables 11 and 12, respectively. These results indicate that for models used in practical analyses of submarine structures, use of the new database alone resulted in over 66% savings in element and stress modulus and an 80% reduction on total run time could be achieved by using the new database and the improvement sparse solver. This corresponded to a speed-up by factor of five. The final collapse mode of the pressure hull and the load-radial displacement curve obtained by VAST are displayed in Figures 18 and 19, respectively. The solutions from all VAST analyses were carefully compared and found to be identical.

For the purpose of identifying the upper limit of the problem size for the present API version of VAST, we considered the full submarine model shown in Figure 20. This model included the entire pressure hull and the major internal structures and contained over 100,000 nodes and over 130,000 elements. For finite element models of this size, the analysis using the original version of VAST completed without any difficulty. However, when the API version was executed with the original sparse solver, the run crashed in the displacement module when the sparse solver tried to allocate memory for the global load vectors. The API VAST run using the improved sparse solver crashed at the decomposition stage due to insufficient memory. This was because that improved sparse solver operates purely in-core at the present time, so it cannot solve problems having more than 70,000 nodes. The run times shown in Table 13 indicated that for the computation modules successfully completed by the improved VAST, very substantial savings were observed for modules that were completed successfully. For instance, the element formulation module was speeded up by a factor of ten.

Table 2: Comparison of run times for test case CS03B

Prefix: CS03B (NL)	Model Size: Node=21, Element=20				
Module	Original VAST	Improved Database only		Improved Database & Solver	
	Time (s)	Time (s)	Red (%)	Time (s)	Red (%)
Element Generation	1.747	1.514	13.337	1.217	30.338
Assembly & Decomposition	0.499	0.818	-63.928	0.452	9.419
Load Vector Generation	0.031	0.022	29.032	0.000	100.000
Displacement Solution	0.780	0.683	12.436	0.624	20.000
Stress Calculation	1.139	0.756	33.626	0.577	49.342
Total	4.649	4.451	4.259	3.323	28.522

Table 3: Comparison of run times for test case CS05B

Prefix: CS05B (NL)	Model Size: Node=91, Element=75				
Module	Original VAST	Improved Database only		Improved Database & Solver	
	Time (s)	Time (s)	Red (%)	Time (s)	Red (%)
Element Generation	4.633	3.326	28.211	3.713	19.858
Assembly & Decomposition	1.045	1.077	-3.062	1.170	-11.962
Load Vector Generation	0.031	0.013	58.065	0.016	48.387
Displacement Solution	1.513	1.032	31.791	1.108	26.768
Stress Calculation	2.356	2.139	9.211	1.888	19.864
Total	10.795	8.642	19.944	8.892	17.629

Table 4: Comparison of run times for test case CS05H

Prefix: CS05H (NL)	Model Size: Node=91, Element=75				
Module	Original VAST	Improved Database only		Improved Database & Solver	
	Time (s)	Time (s)	Red (%)	Time (s)	Red (%)
Element Generation	8.725	5.582	36.023	6.162	29.375
Assembly & Decomposition	2.017	1.717	14.874	1.825	9.519
Load Vector Generation	0.023	0.008	65.217	0.000	100.000
Displacement Solution	2.606	1.481	43.170	1.934	25.787
Stress Calculation	5.966	4.457	25.293	5.070	15.018
Total	21.222	14.837	30.087	16.645	21.567

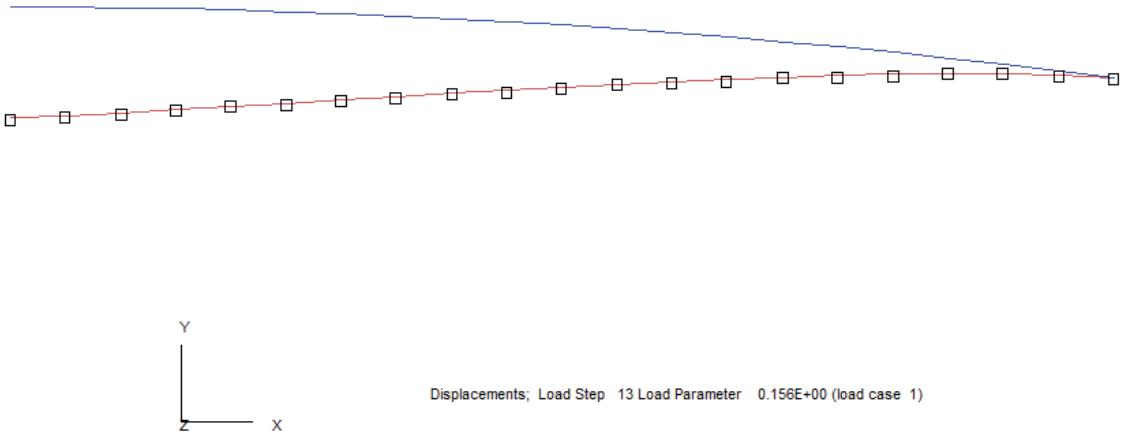

Figure 3: Original and deformed configurations for test case CS03B which involved snap-through of a simply supported shallow arch subjected to a centre point load

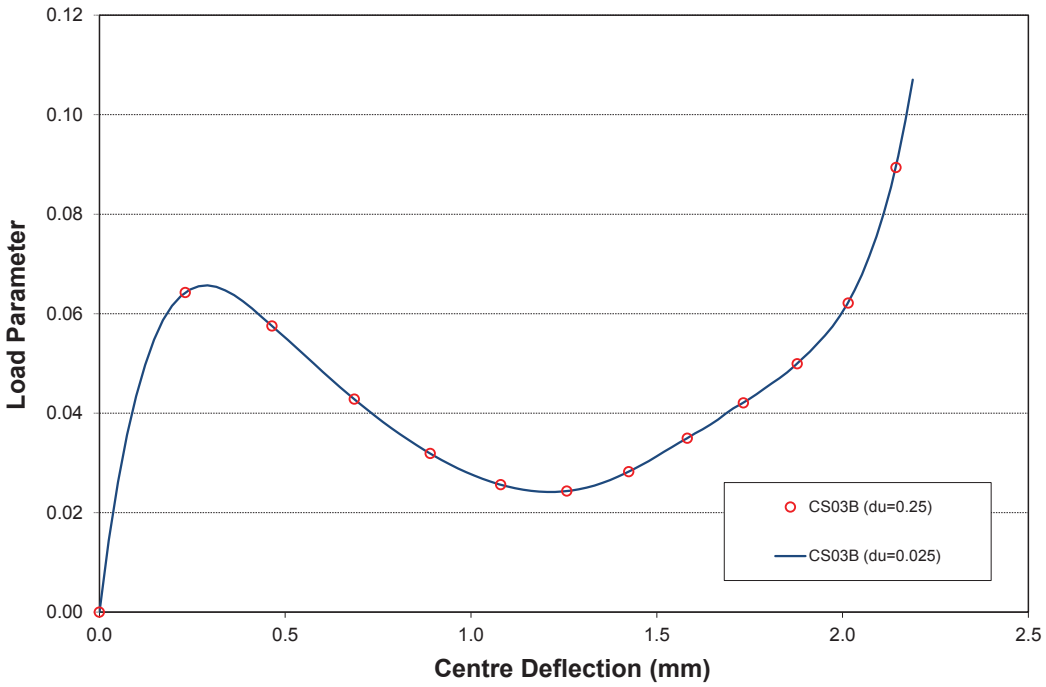

Figure 4: Load-centre deflection curves obtained for test case CS03B using large and small solution steps

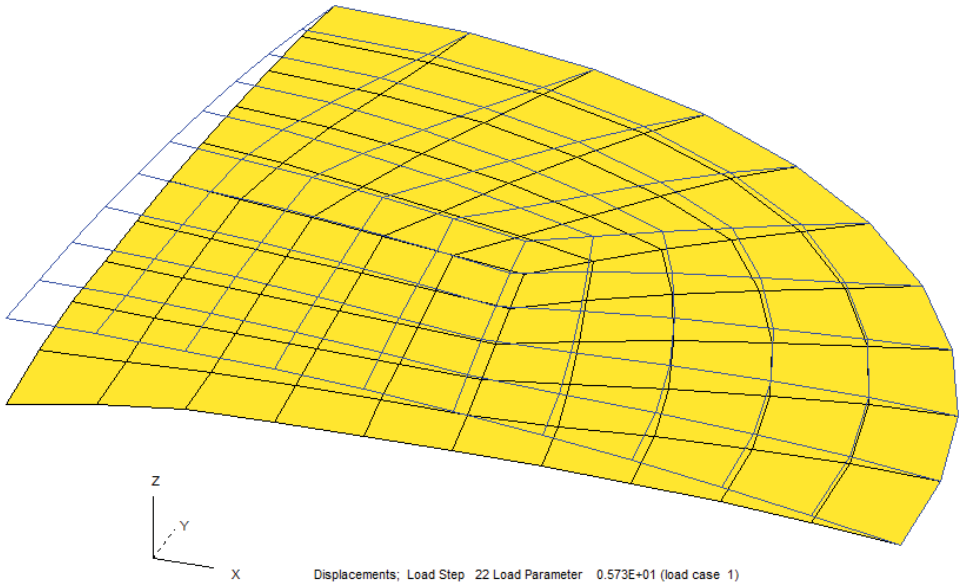

Figure 5: Original and deformed configurations for test case CS05B which involved a hinged shallow spherical shell subjected to a centre point load

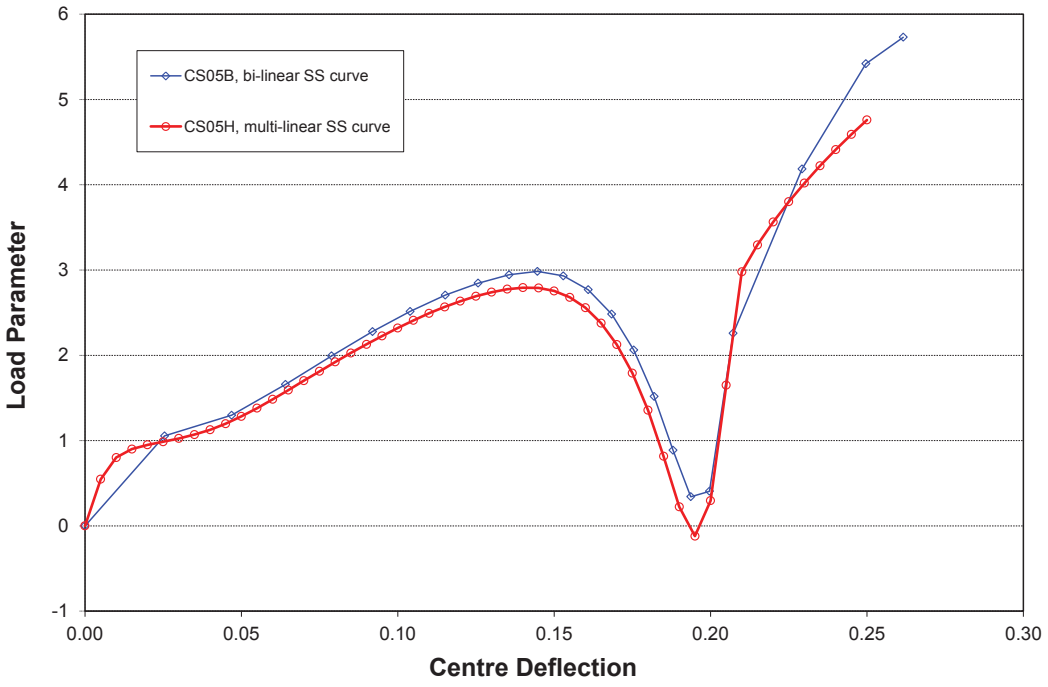

Figure 6: Load-centre deflection curves obtained for test cases CS05B and CS05H which involved bi-linear and multi-linear stress-strain behaviours, respectively

Table 5: Comparison of run times for coarse FE model of stiffened panel FB6

Prefx: panel_FB6 (L)	Model Size: Node=3484, Element=3366				
Module	Original VAST	Improved Database only		Improved Database & Solver	
	Time (s)	Time (s)	Red (%)	Time (s)	Red (%)
Element Generation	0.515	0.250	51.456	0.250	51.456
Assembly & Decomposition	1.186	1.139	3.963	0.577	51.349
Load Vector Generation	0.484	0.016	96.694	0.016	96.694
Displacement Solution	0.343	0.296	13.703	0.390	-13.703
Stress Calculation	0.390	0.343	12.051	0.343	12.051
Total	2.948	2.044	30.665	1.591	46.031

Table 6: Comparison of run times for refined FE model of panel FB6

Prefix: panel_FB6-f1 (L)	Model Size: Node=13699, Element=13464				
Module	Original VAST	Improved Database only		Improved Database & Solver	
	Time (s)	Time (s)	Red (%)	Time (s)	Red (%)
Element Generation	2.028	0.874	56.903	0.874	56.903
Assembly & Decomposition	7.675	7.129	7.114	2.590	66.254
Load Vector Generation	1.966	0.047	97.609	0.047	97.609
Displacement Solution	1.498	1.201	19.826	1.435	4.206
Stress Calculation	1.544	1.513	2.008	1.357	12.111
Total	14.726	10.780	26.796	6.334	56.988

Table 7: Comparison of run times for further refined FE model of panel FB6

Prefix: panel_FB6-f2 (L)	Model Size: Node=54325, Element=53856				
Module	Original VAST	Improved Database only		Improved Database & Solver	
	Time (s)	Time (s)	Red (%)	Time (s)	Red (%)
Element Generation	9.001	3.463	61.526	3.463	61.526
Assembly & Decomposition	52.089	50.591	2.876	11.107	78.677
Load Vector Generation	8.642	0.172	98.010	0.187	97.836
Displacement Solution	5.928	5.616	5.263	6.271	-5.786
Stress Calculation	6.256	5.382	13.971	5.460	12.724
Total	82.025	65.271	20.425	26.536	67.649

Table 8: Comparison of run times for nonlinear collapse analysis of panel FB6 subjected to compression in the transverse direction

Prefix: panel_FB6 (NL)	Model Size: Node=3484, Element=3366				
Module	Original VAST	Improved Database only		Improved Database & Solver	
	Time (s)	Time (s)	Red (%)	Time (s)	Red (%)
Element Generation	224.168	147.473	34.213	146.738	34.541
Assembly & Decomposition	362.429	330.496	8.811	152.638	57.885
Load Vector Generation	0.500	0.016	96.800	0.018	96.400
Displacement Solution	24.257	11.285	53.477	13.866	42.837
Stress Calculation	156.696	68.116	56.530	78.559	49.865
Total	786.436	560.869	28.682	395.046	49.768

Table 9: Comparison of run times for nonlinear collapse analysis of panel L10 subjected to compression in the axial direction

Prefix: panel_01_NL (NL)	Model Size: Node=4503, Element=4368				
Module	Original VAST	Improved Database only		Improved Database & Solver	
	Time (s)	Time (s)	Red (%)	Time (s)	Red (%)
Element Generation	323.356	189.698	41.335	203.186	37.163
Assembly & Decomposition	583.432	513.854	11.926	242.086	58.507
Load Vector Generation	0.060	0.001	98.333	0.016	73.333
Displacement Solution	25.861	12.737	50.748	19.021	26.449
Stress Calculation	231.530	116.070	49.868	128.755	44.389
Total	1191.619	836.059	29.838	596.445	49.947

Table 10: Comparison of run times for nonlinear collapse analysis of panel L10 subjected to combined normal pressure and transverse compression

Prefix: panel_01_2LC	Model Size: Node=4503, Element=4368				
Module	Original VAST	Improved Database only		Improved Database & Solver	
	Time (s)	Time (s)	Red (%)	Time (s)	Red (%)
Element Generation	239.638	152.220	36.479	148.306	38.112
Assembly & Decomposition	457.505	400.314	12.501	161.739	64.648
Load Vector Generation	0.800	0.018	97.750	0.016	98.000
Displacement Solution	25.299	10.814	57.255	13.493	46.666
Stress Calculation	146.893	69.689	52.558	78.238	46.738
Total	890.375	635.769	28.595	404.216	54.602

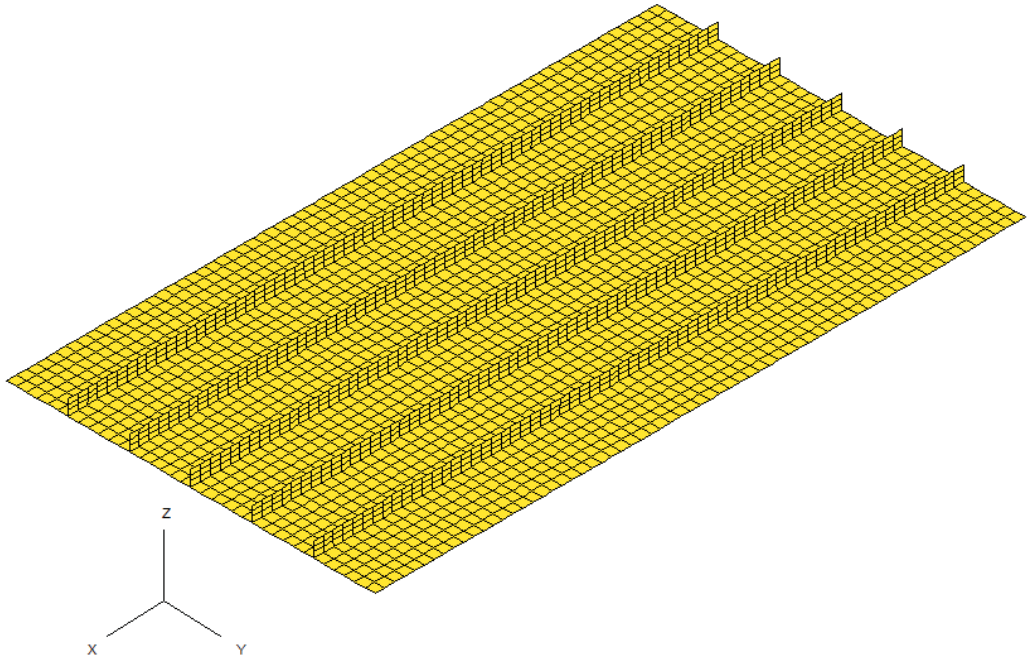

Figure 7: Coarse finite element model of panel FB6

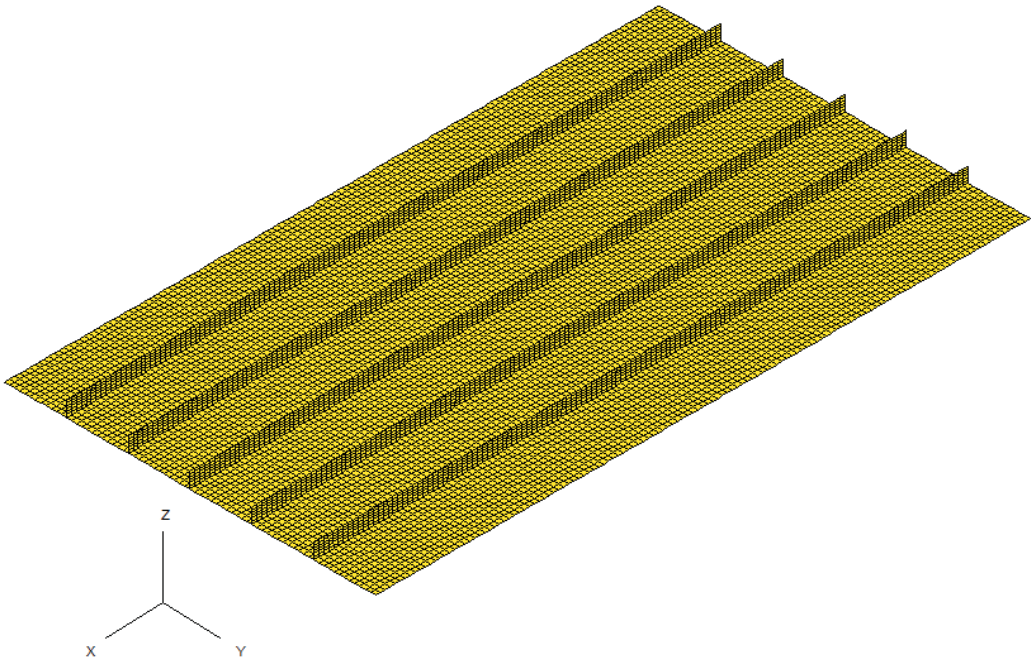

Figure 8: Refined finite element model of panel FB6

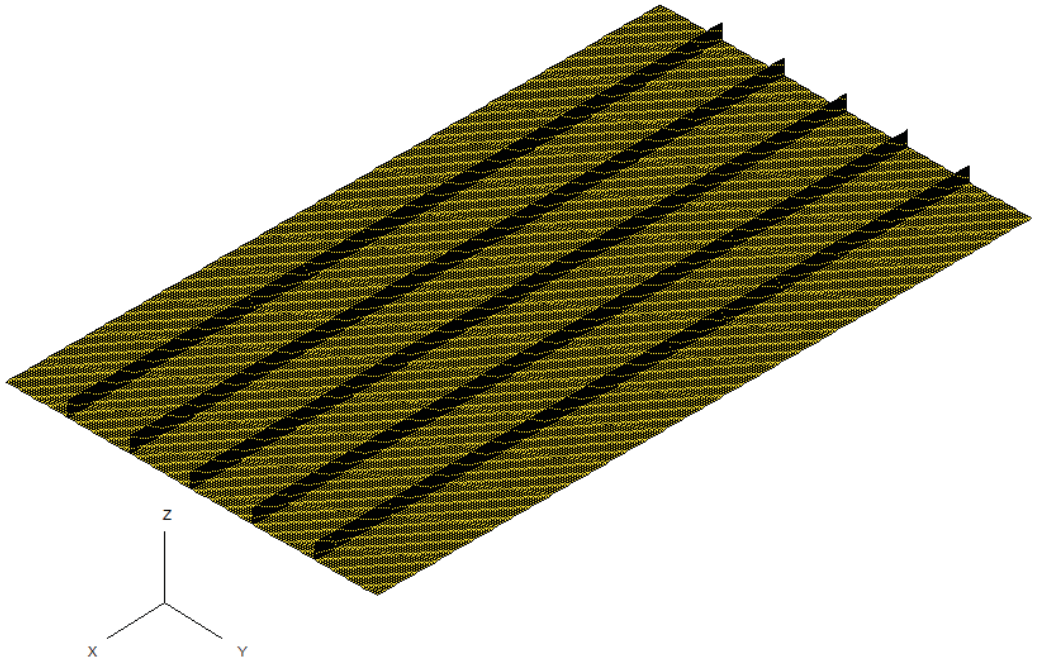

Figure 9: Further refined finite element model of panel FB6

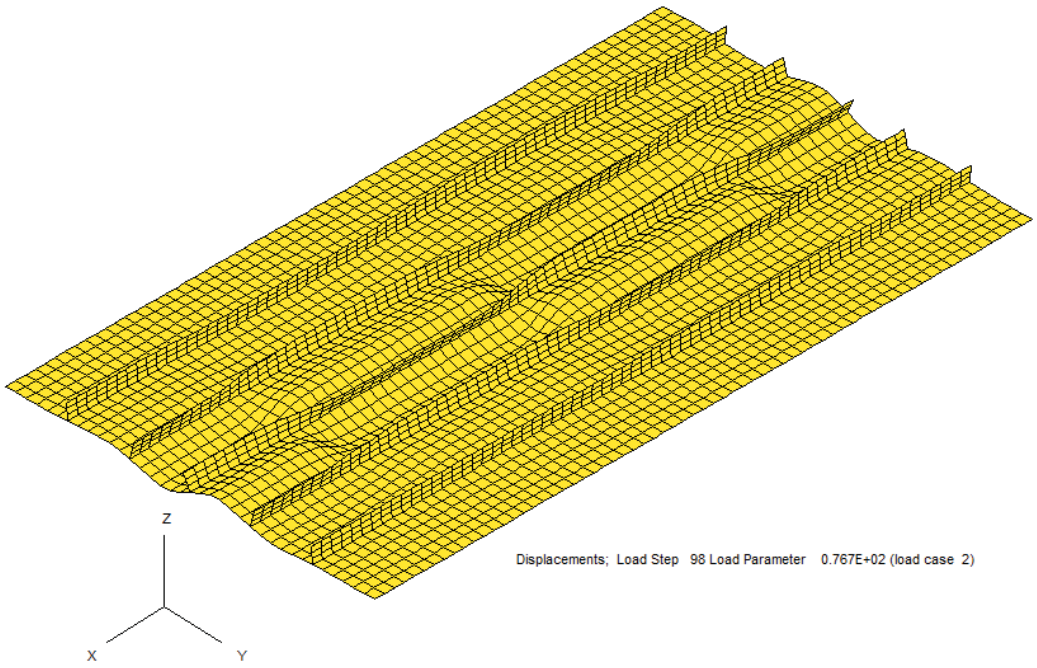

Figure 10: Final collapse mode of panel FB6 subjected to transverse compression

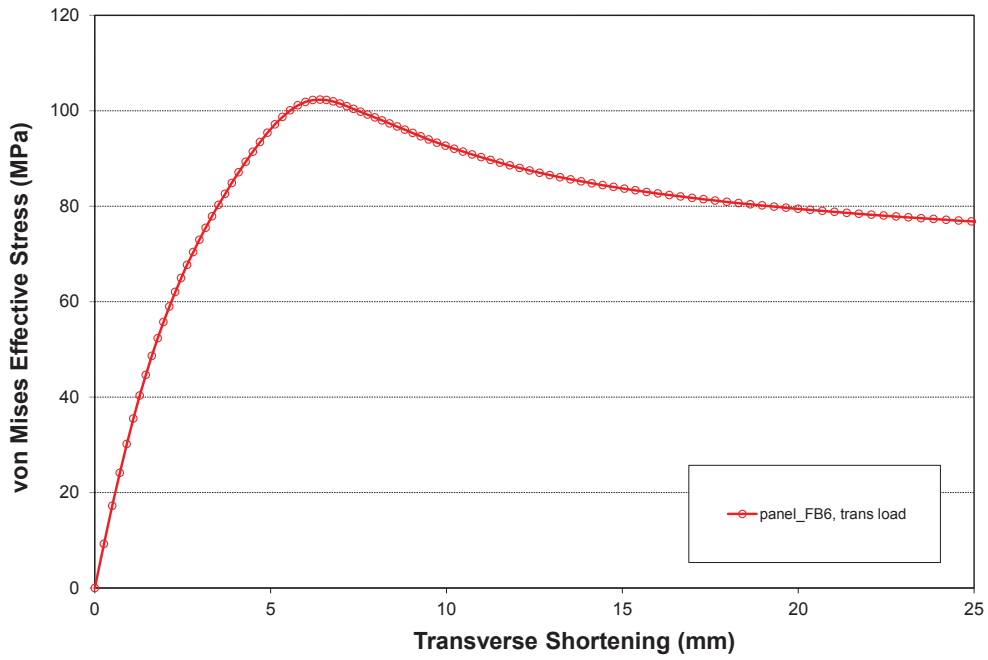

Figure 11: Transverse load-shortening curve of panel FB6

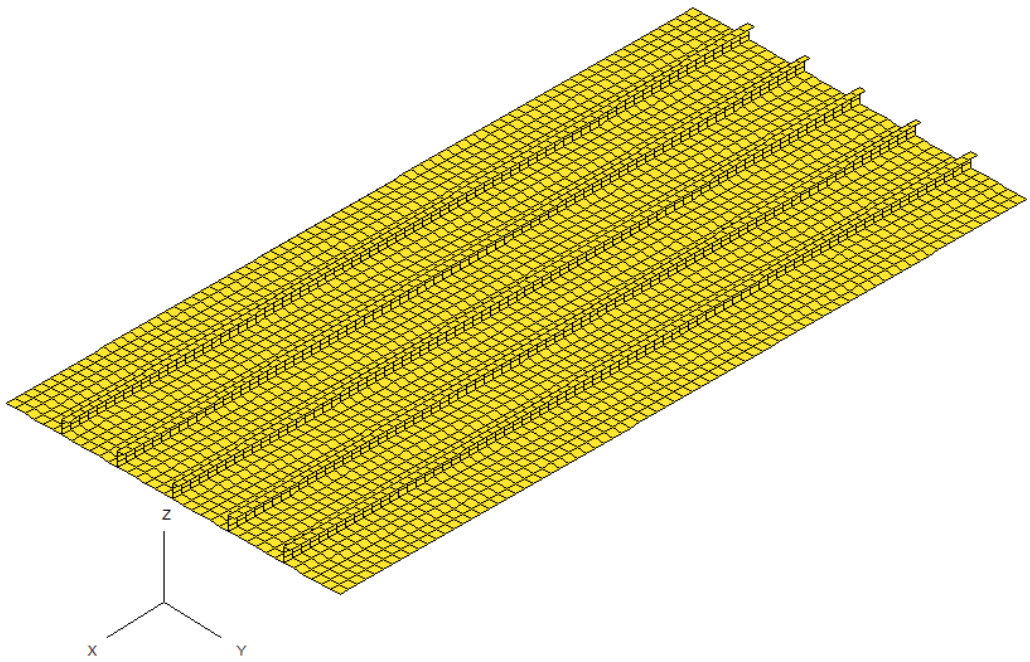

Figure 12: Original un-deformed finite element model of panel L10

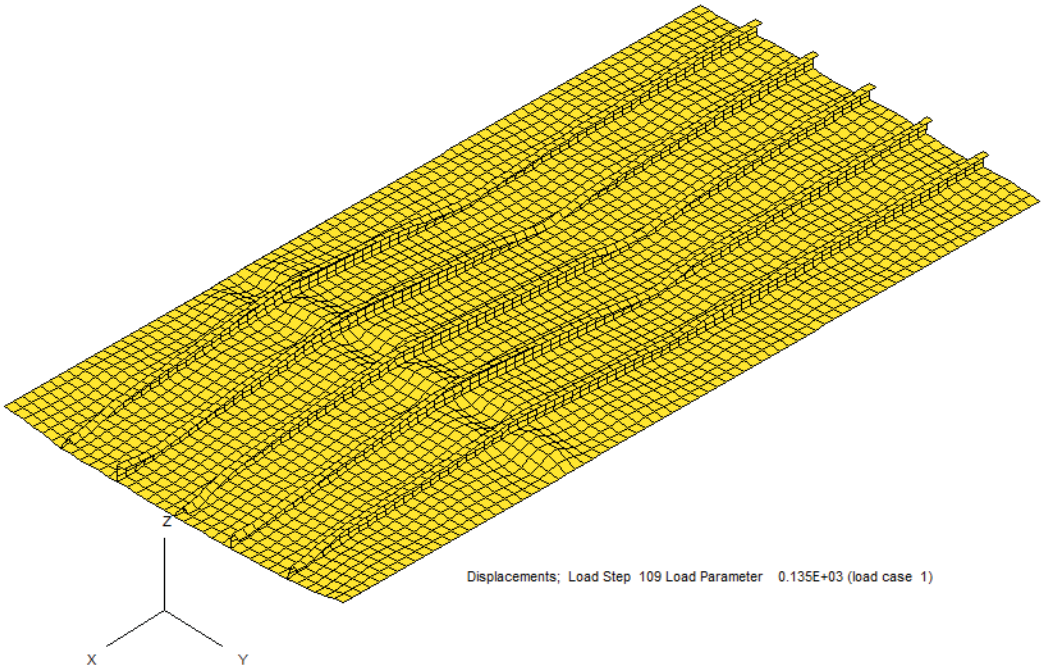

Figure 13: Final collapse mode of panel L10 subjected to axial compression

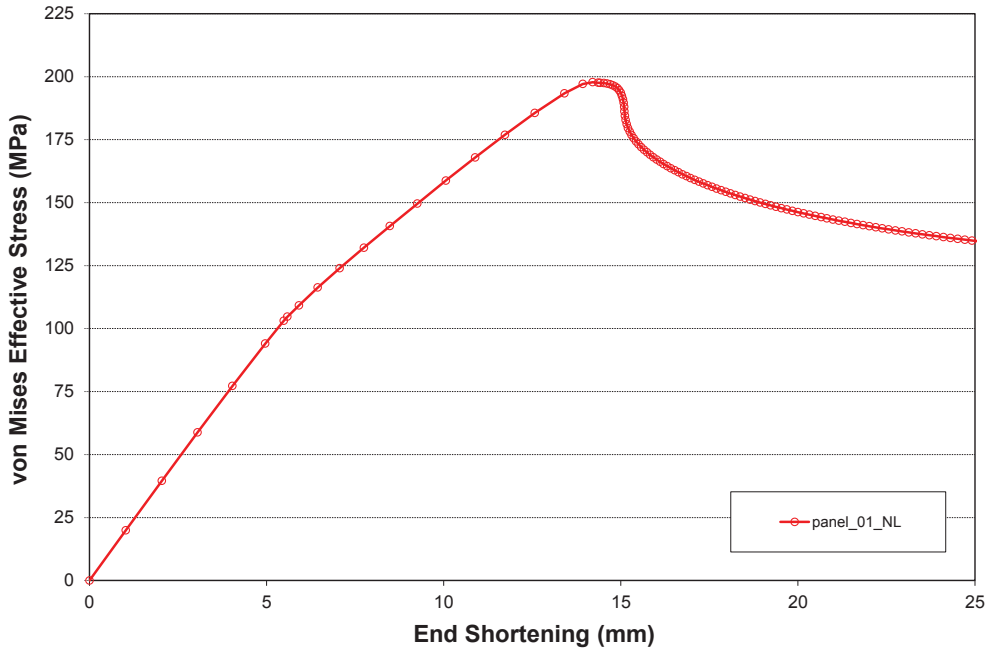

Figure 14: Load-shortening curve of panel L10 subjected to axial load

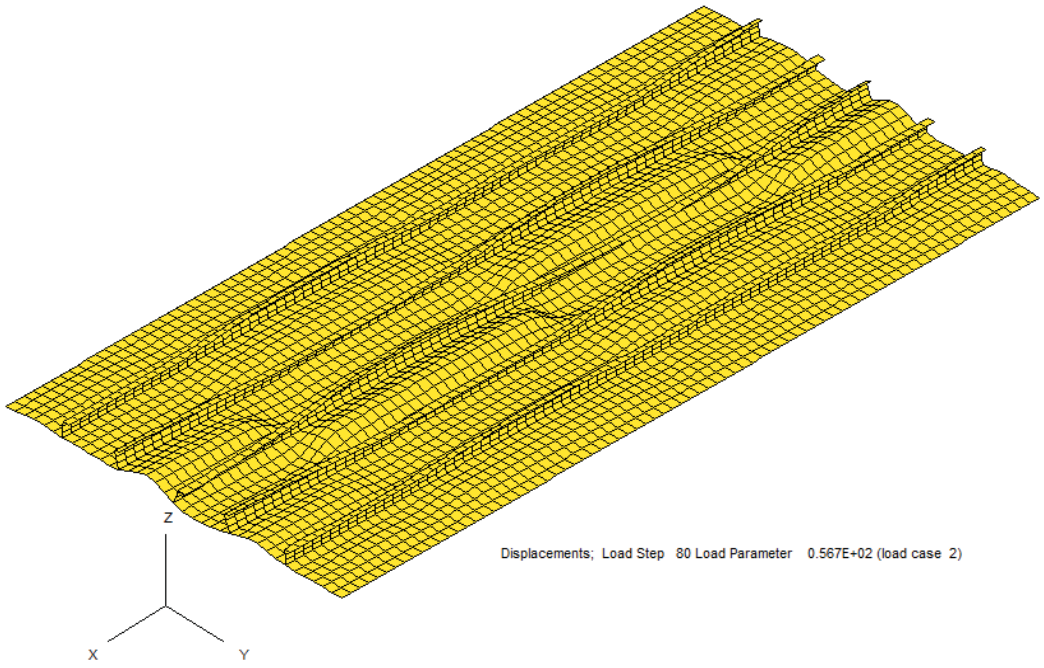

Figure 15: Final collapse model of panel L10 subjected to combined normal pressure and transverse compression

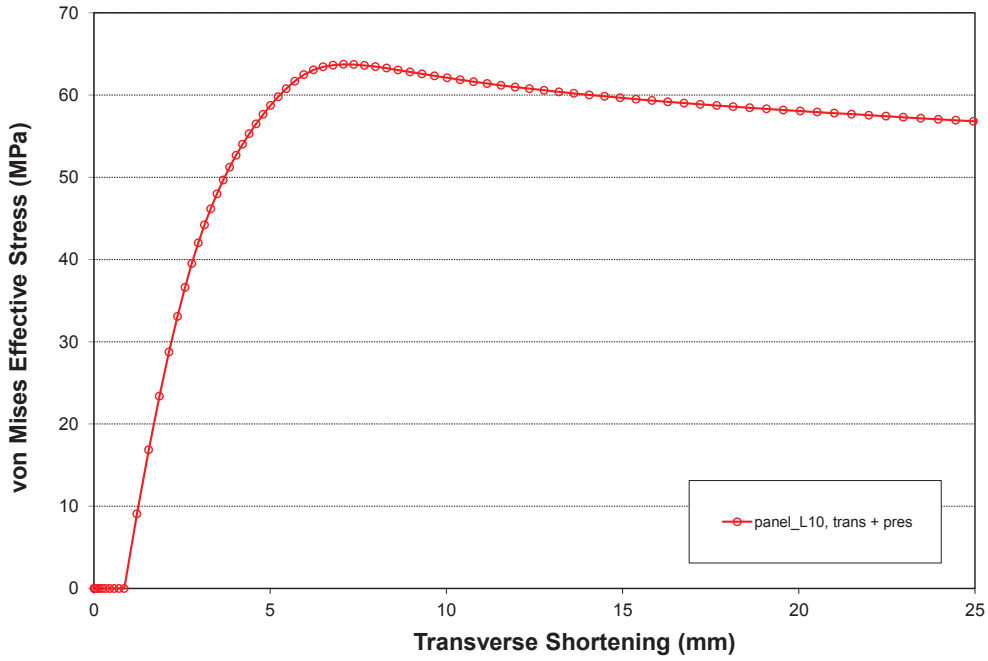

Figure 16: Transverse load-shortening curve obtained for combined pressure and transverse compression. The initial displacement was due to the pressure load

Table 11: Comparison of run times for linear analysis of a submarine compartment model

Prefix: Comp2_hull_lin (L)	Model Size: Node=20965, Element=24281				
Module	Original VAST	Improved Database only		Improved Database & Solver	
	Time (s)	Time (s)	Red (%)	Time (s)	Red (%)
Element Generation	3.274	1.240	62.126	1.248	61.881
Assembly & Decomposition	41.302	40.340	2.329	6.833	83.456
Load Vector Generation	1.623	0.035	97.843	0.047	97.104
Displacement Solution	1.220	0.951	22.049	1.061	13.033
Stress Calculation	1.395	0.927	33.548	1.186	14.982
Total	48.851	43.524	10.905	10.405	78.701

Table 12: Comparison of run times for nonlinear collapse analysis of the submarine model

Prefix: Comp2_hull (NL)	Model Size: Node=20965, Element=24281				
Module	Original VAST	Improved Database only		Improved Database & Solver	
	Time (s)	Time (s)	Red (%)	Time (s)	Red (%)
Element Generation	2375.122	807.289	66.011	796.111	66.481
Assembly & Decomposition	11737.926	11575.470	1.384	1789.394	84.755
Load Vector Generation	1.716	0.034	98.019	0.047	97.261
Displacement Solution	135.345	93.140	31.183	90.904	32.835
Stress Calculation	2030.488	667.721	67.115	664.836	67.257
Total	16382.465	13148.370	19.741	3344.927	79.582

Table 13: Comparison of run times for linear analysis of a full submarine model

Prefix: Comp1234_LS (L)	Model Size: Node=101893, Element=133529				
Module	Original VAST	Improved Database only		Improved Database & Solver	
	Time (s)	Time (s)	Red (%)	Time (s)	Red (%)
Element Generation	55.782	5.725	89.737	5.567	90.020
Assembly & Decomposition	344.468	345.306	-0.243	Stopped	
Load Vector Generation	10.233	2.231	78.198		
Displacement Solution	3.994	Stopped			
Stress Calculation	6.100				
Total	421.295				

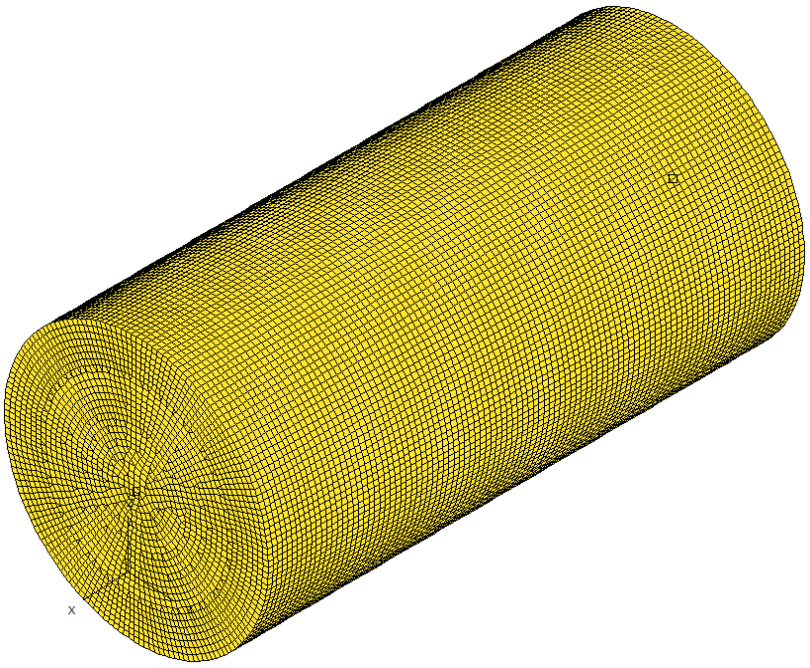

Figure 17: Original un-deformed finite element model of a submarine compartment

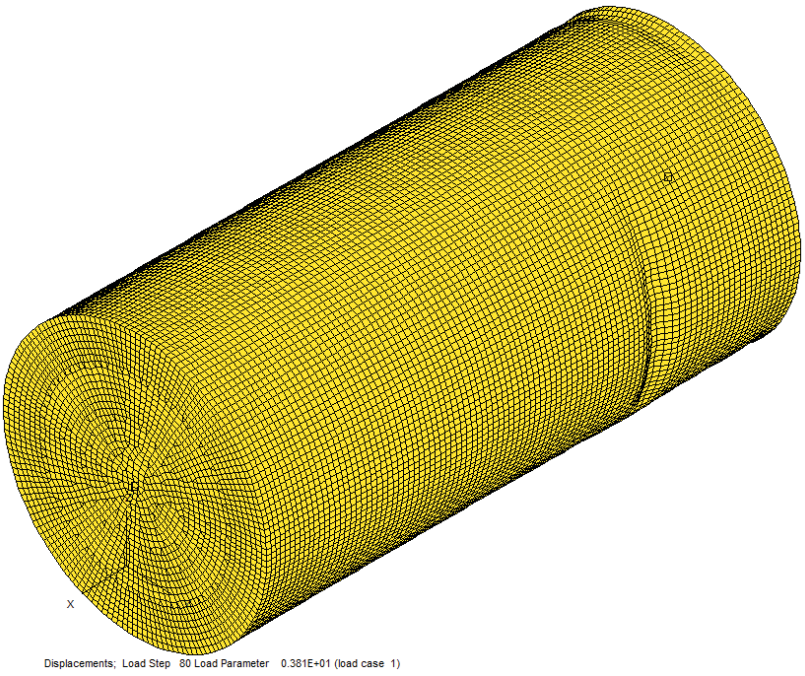

Figure 18: Final collapse mode of the submarine compartment model

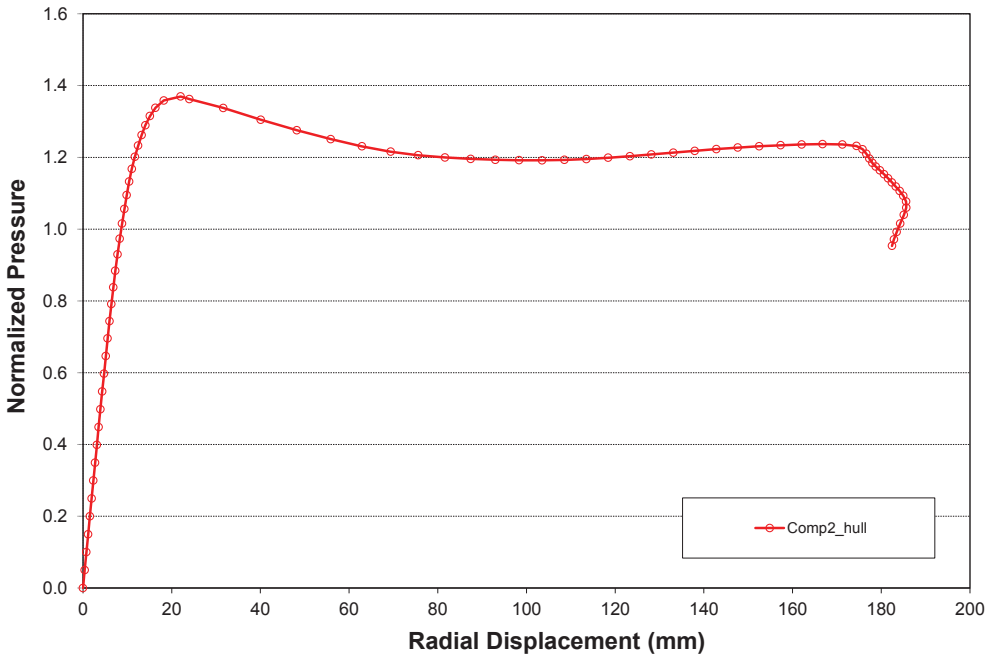

Figure 19: Load-displacement curve at a node in the middle of the buckled area

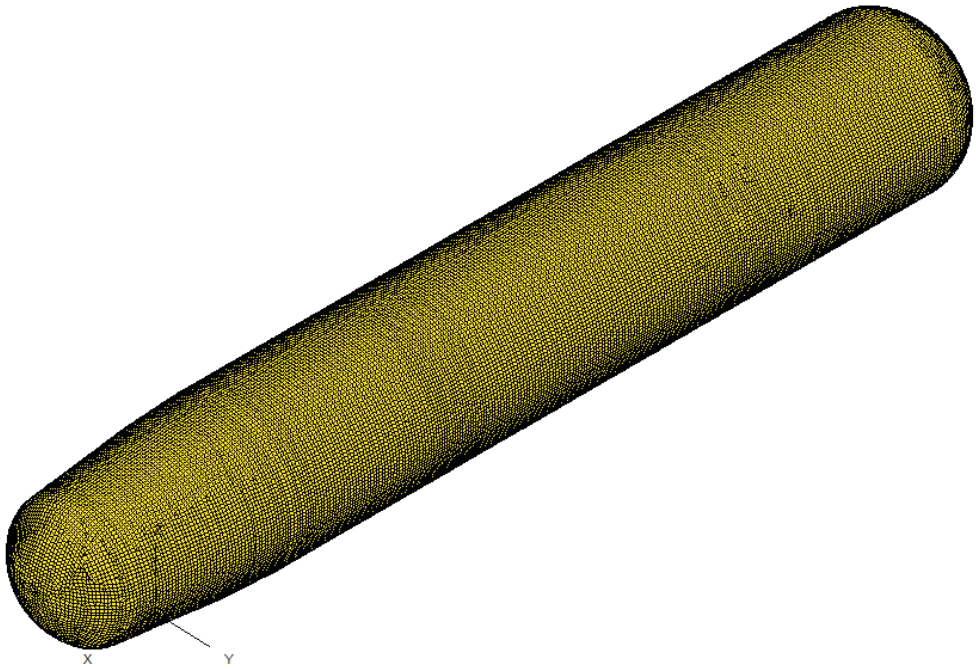

Figure 20: Full submarine model including pressure hull and internal structures

4 Conclusions

The report described recent development and implementation of an in-core database for VAST. The database was developed based on the C++ Standard Template Library (STL) generic data structures and contained a set of Application Programming Interface (API) functions that allowed the database to be accessed from a Fortran program, like VAST. In order to minimize the number of the API functions and to allow easy expansion of the database in the future, the arguments of the API functions have been designed to simply contain integer and real arrays, rather than more specific data types related to certain element formulations or material/geometric properties.

Implementation of the database in VAST has required significant restructuring of the program. In particular, a pre-processor module, named PREPR1, was developed to read all the input data and store them into the database through the use of the API functions. During the VAST calculations, all the finite element model information, such as nodes, elements, material/geometric properties, boundary conditions and loads, and the intermediate results, such as the current displacement and stresses, are repeatedly stored and retrieved to and from the database. This treatment of data flow eliminated a large amount of I/O operations in the original version of VAST and resulted in a large savings on computation time. In order to demonstrate the full potential for speeding up VAST, the newly improved sparse solver was also incorporated.

At the present time, this new version of VAST contained two most commonly used elements in VAST, namely 2-noded general beam and 4-noded quad shell elements. However, all linear and nonlinear static analyses options were operational. It has been verified and benchmarked using test problems of different sizes, ranging from small test models taken from the standard VAST Autotester to relatively large models previously utilized for practical nonlinear collapse analyses of submarine pressure hull. The benchmark results indicated that by combining the new database and the new sparse solver, the overall speed of VAST was increased by a factor of five for practical engineering problems.

This work clearly demonstrates the potential for making VAST highly efficient by using a database and improved sparse solver. However, the current API version of VAST only provides limited capabilities and is capable of solving relatively small problems because both the database and the new sparse solver operated purely in memory on a 32-bit operating system. Significant efforts are still required to generate a fully functional version of VAST. These may include efforts for exploring the methods for further efficiency improvements, removing the limit on model size, and to providing all computational capabilities currently supported by VAST. The current API version of VAST is ideal for being used as a test bed for many of these investigations.

In order to achieve this goal, the following tasks are recommended:

- Expand the database to include out-of-core processing using the Hierarchical Data Format version 5 (HDF5) or other suitable technology and benchmark the performance.
- Parallelize element formulation, global matrix assembly and stress calculation using OpenMP.
- Complete the improved sparse solver by expanding it to 64-bit, providing an out-of-core option and speeding up forward reduction and backward substitution.

References

[1] VAST User's Manual, Version 9.1. Martec Limited, Halifax, 2012.

[2] Link, R. "Parallelization of Sparse Solver - Phase 1", Martec Technical Report TR-12-17, Martec Limited, Halifax, May 2012.

[3] Link, R. "VAST Parallel Sparse Solver", Martec Technical Report TR-13-39, Martec Limited, Halifax, August 2013.

[4] Stepanov, A. and Lee, M., The Standard Template Library. HP Laboratories Technical Report 95-11(R.1), November 1995.

[5] The eXtensible Data Model and Format (XDMF). http://www.xdmf.org

[6] Tautges, T. J. et. al., MOAB: A Mesh-Oriented Database, Sandia National Laboratories Report SAND2004-1592, April 2004.

[7] The HDF Group. Hierarchical data format version 5, 2000-2010. http://www.hdfgroup.org/HDF5.

[8] Doyle, C., "HDF5 File System for the VAST API", Martec Technical Report TN-13-40, Martec Limited, Halifax, September 2013.

[9] Jiang, L. Trident FEA 2011 – List of Test Cases in Autotester for VAST91, Martec Limited. Halifax, March 2011.

[10] Jiang, L., Link, R. and Wallace, J. "Nonlinear Finite Element Collapse Analyses of Stiffened Panels", Martec Technical Report TR-11-02, February, 2011.

[11] Jiang, L., "Nonlinear Finite Element Collapse Analyses of Stiffened Panels - Phase 2", Martec Technical Report TR-11-10, March, 2011.

[12] Tobin, S., Wallace, J., Jiang, L. MacAdam, T. and Norwood, M. "Submarine Structure Modeling and Analysis for Life-Cycle Management - Phase 3", DRDC CR 2008-085, Martec Limited, Halifax, June 2008.