NASA Technical Memorandum 100642

The CSM Testbed Software System:
A Development Environment for Structural Analysis Methods on the NAS CRAY-2

(NASA-TM-100642) THE CSM TESTBED SOFTWARE SYSTEM: A DEVELOPMENT ENVIRONMENT FOR STRUCTURAL ANALYSIS METHODS ON THE NAS CRAY-2 (NASA) 46 p CSCL 20K

N89-11287

Unclas

G3/39 0170002

Ronnie E. Gillian and Christine G. Lotts

September 1988

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Table of Contents

The CSM Testbed Software System:
A Development Environment for Structural Analysis Methods on the NAS CRAY-2

THE CSM TESTBED SOFTWARE SYSTEM: A Development Environment for Structural Analysis Methods on the NAS CRAY-2

INTRODUCTION

The Computational Structural Mechanics (CSM) Activity at the NASA Langley Research Center is developing structural analysis methods that exploit modern computers (ref. 1). To facilitate that research effort, a development environment has been constructed to insulate the researcher from the many computer operating systems of a widely distributed computer network. This paper describes that environment and its extension to include the supercomputer resources of the Numerical Aerodynamic Simulator (NAS) CRAY-2^{TM}, at the NASA Ames Research Center.

The field of computational structural analysis is dominated by two types of computer programs. One type is the huge, 2000 subroutine (ref. 2), general purpose program that is the result of over a hundred man-years of effort spanning over a decade; the other type is the relatively small code resulting from an academic or research environment that represents a one- to two-year effort for a specific research application. This dichotomy has resulted in long delays in making research technology available for the critical structures problems that NASA faces. To address the problem of accelerating the introduction of successful research technology into large-scale applications programs, a modular, public–domain, machine independent, architecturally simple, software development environment, denoted the CSM Testbed, is being constructed.

A development environment which insulates both the structural analyst using the Testbed and the methods developer writing enhancements for it is important in a distributed environment. Distributed environments are made up of stand-alone computers of different sizes, architectures, and vendors, with a common network protocol offering the user easy file transfer and remote login functions. Structural analysts require the diverse computer capabilities offered by a distributed environment (workstation–mainframe–supercomputer), but cannot afford the "overhead" of learning the operating system commands for each system they use. Methods developers have a similar problem, but at a lower level. They cannot afford the "overhead" of learning a new set of system calls for each computer on which they wish to implement their application code. The CSM Testbed addresses these problems.

The CSM Testbed development environment was ported to the CRAY-2 to provide a high end computational capability for structural analysis research. Earlier Testbed development efforts were directed toward mainframes and minicomputers even though the complexity of the structures that were being analyzed was growing. It was only through the structural analyses required during the space shuttle Challenger accident investigation and subsequent recertification program (ref. 3) that the magnitude of the computational task required for large-scale structural analyses was fully appreciated. After that experience, it was clear that if this research Testbed was to be used to learn how to solve problems of critical interest to NASA, the Testbed would have to be available on a true supercomputer. To that end, the CSM Testbed was ported to the NAS CRAY-2 at the Ames Research Center. This paper describes the implementation experiences, the resulting capability, and the future directions for the Testbed on supercomputers.

PROGRAM STRUCTURE

The CSM Testbed program is an example of a modern software architecture designed to support development of engineering analysis methods as well as to perform engineering analyses. Its organization is illustrated in figure 1. The inner circle in figure 1, the computer operating system, is provided by the computer vendor and is different for each vendor. The outer ring in the figure, the development environment, insulates both the user and the methods developer from those differences by providing a consistent interface.

The CSM Testbed is written primarily in Fortran and is organized as a single executable file, called a macroprocessor. The macroprocessor calls structural applications modules (also known as processors) that have been incorporated as subroutines. Applications modules are installed into the macroprocessor as they become accepted in the structural analysis community and are of a general interest to other researchers. The macroprocessor and applications modules interface with the operating system for their command input and data management functions through a set of "architectural utilities" that originated in a software system called NICE (Network for Integrated Computational Elements) (ref. 4). Processors access the Testbed utilities by calling entry points implemented as Fortran-77 functions and subroutines which are available to module developers in the Testbed object libraries. Applications modules do not communicate directly with each other, but instead communicate by exchanging named data objects in a database managed by a data manager called GAL (Global Access Library). The user controls execution of applications modules via an interactive or batch command input stream written in a command language, called CLAMP (Command Language for Applied Mechanics Processors) which is processed by CLIP, the Command Language Interpreter Program. Command language procedures for performing complex analysis tasks may be developed and stored for future use.

To facilitate the development of new methods and algorithms, a capability for independent executable programs to perform special functions related to a Testbed analysis is included. Applications may be developed independently, using the Testbed architectural utilities and data management capabilities, and may be invoked from within a Testbed command procedure or runstream; this type of program is called a Testbed external processor. The macroprocessor and external processors may be used within a single Testbed command input stream via the architectural utility, SuperCLIP (ref. 5), which performs the interprocessor exchange via operating system calls which are invisible to the user. This capability provides much of the flexibility required of the CSM Testbed as it relates to the development of new applications.

For structural analysis research, both interactive and batch modes of operation are required and both are supported in this program. Throughout the CSM Testbed development effort, attention has been given to the problems associated with research codes and their requirements for generality, while keeping a watch on overall efficiency. Efficiency affects the overall size and complexity of the structural problems that can be considered, and if current structures research problems are to be solved, efficiency must be maintained.

<u>The Testbed Command Language</u>

The Testbed command language, called CLAMP, is a generic language originally designed to support the NICE system and to offer program developers the means for building problem-oriented languages (ref. 6). It may be viewed as a stream of free-field command records read from an appropriate command source (the user terminal, actual files, or processor messages). The commands are interpreted by a "filter" utility called CLIP, whose function is to produce object records for consumption by its user program. The standard operating mode of CLIP is the processor-command mode. Commands are directly supplied by the user, retrieved from ordinary card-image files, or extracted from the global database, and submitted to the running processor. Special commands, called directives, are processed directly by CLIP; the processor is "out of the loop". Transition from processor-command to directive mode is automatic. Once the directive is processed, CLIP returns to processor-command mode. Directives are used to dynamically change run-environment parameters, to process advanced language constructs such as macrosymbols and command procedures, to implement branching and cycling, and to request services of the data manager. CLIP can be used in this way to provide data to a processor as well as to control the logic flow of the program through a single input stream. All command language directives are available to any processor that uses the CLIP-Processor interface entry points.

Program execution begins with control transferred to the Testbed macroprocessor by the computer operating system. Within the macroprocessor an entry point requesting a read operation is called which begins parsing and interpreting command language directives. This operation continues until a complete processor command record has been read and processed by CLIP. That data record is then returned to the calling module to be interpreted according to the application or macroprocessor originally requesting the read operation. The next read operation continues the cycle until a command is encountered which directs the program to terminate. This process not only provides for module sequence control, but also provides a powerful input data description language for preparing processor input. Both capabilities are incorporated through the same applications language environment.

The NICE Data Manager

The data manager within the CSM Testbed was derived from the Global Access Library (GAL) concept developed at the Lockheed Palo Alto Research Laboratory (ref. 7). Methods for data management in structural analysis programs can be broken into three levels of complexity: file systems, file partition systems, and data base systems (ref. 8). Since GAL database files are subdivided or partitioned into data sets the Testbed data manager is classified as a file partition manager. To a processor, a GAL data library is analogous to a file. It must be opened, written, read, closed, and deleted explicitly. The global access library resides on a direct-access disk file and contains a directory structure called a table of contents (TOC) through which specific data sets may be addressed. Low level routines access the GAL library file in a word-addressable scheme as described by Felippa in reference 9. The data management system is accessible to the user through the command language directives and to the running processors through the GAL-Processor interface.

The actual I/O interface to the UNICOS^{TM} operating system for the CRAY-2 is accomplished through a set of block I/O routines written in the C programming language. To provide the efficiency required to process the volume of data required for a complex structural analysis, all usual overhead associated with Fortran has been eliminated. Listings of the routines used for block I/O in the Testbed are presented in Appendix A.

The global database is made up of sets of data libraries (GALs) residing on direct-access disk files. Data libraries are collections of named datasets, which are collections of dataset records. The data library format supported by the Testbed is called GAL/82, which can contain nominal datasets made up of named records. Some of the advantages to using this form of data library are: 1) the order in which records are defined is irrelevant, 2) the data contained in the records may be accessed from the command level, and 3) the record data type is maintained by the manager; this simplifies context-directed display operations and automatic type conversion.

SuperCLIP Implementation

The SuperCLIP capability of the Testbed architecture performs *inter*processor control, allowing independent programs which use the Testbed architecture facilities (CLIP and GAL) to be executed from within a single Testbed input stream. SuperCLIP handles the interprocessor CLIP state preservation and restoration so that the CLIP environment is maintained across independent program executions. These independent programs can be used in conjunction with the Testbed macroprocessor, other independent Testbed processors, or entirely alone, as appropriate to accomplish the required task. The implementation of SuperCLIP is the most complex and machine dependent element of the Testbed architecture software. To date it has been implemented under VAX^{TM}/VMS^{TM} and the UNIX^{TM} operating systems.

The operations performed by SuperCLIP to accomplish the processor exchange are as follows:

1. The name of the executable file for the new processor is pushed onto a stack data structure called the Process Name Stack.
2. The CLIP data structures are saved. This is done by writing the contents of the CLIP data structures to a file, named ZZZZZZZ, via the data manager. The structures include the Decoded Item Table, Macrosymbol Table, Command Source Stack, Process Name Stack, control characters, logical unit table, and list of active data libraries. All

open libraries are closed.

3. The process switch is performed. For the VAX/VMS version, this is done via the LIB$RUN_PROGRAM system function. For UNIX, it is done via the EXECLP system function. Both functions stop the current process and start the target process.
4. CLIP is initialized in the new processor. The new processor calls CLIP for data input; CLIP tests for the existence of the ZZZZZZZ file to determine if this processor is executing as the result of a SuperCLIP operation. If the file exists, the CLIP state is restored.
5. The CLIP state is restored. The ZZZZZZZ file is read via the data manager to restore the CLIP data structures. Non-scratch libraries that were open in the parent processor are re-opened. The Command Source Stack is reconstructed so it has the same array of open files, and script files are restored to their original positions. The ZZZZZZZ file is closed with the delete option so it disappears.

A Testbed processor terminates via a SuperCLIP function which performs many of the same functions described above. The first step is the only one which is different. Here the name of the parent processor is extracted from the Process Name Stack, which is then removed from the stack. If the stack is empty, a normal termination is performed. If the stack is not empty, the parent processor becomes the target process and steps 2-5 above are performed

The only part of the SuperCLIP operation which is different for the two implementations is the process switch operation which requires different system function calls for the different systems. In the UNIX version, any files which were opened via the Fortran OPEN statement must also be closed, since files are allowed to remain open across the EXECLP calls.

Structural Application Modules

The application modules are installed in the Testbed system in a macroprocessor configuration. They perform functions related to a structural analysis task, including model definition, element interconnection analysis, system matrix assembly and factoring, static stress analysis, eigenvalue analysis, thermal analysis, data display, and postprocessing functions. Additional pre- and post-processing functions have been implemented as independent executable programs called external processors. The initial structural analysis functions were implemented by interfacing processors from the SPAR structural analysis system with the NICE architecture utilities (ref. 10).

Since the initial installation, many new modules have been developed to replace or complement the original functions. A software "shell" and utilities have been developed to facilitate installation of new types of structural elements into the Testbed. Several element processors have been developed based on this shell and have been installed in the Testbed. Pre- and post-processing functions as well as modules implementing new solution algorithms have also been developed and installed. A description of the current analysis capability of the Testbed is presented in reference 11.

The architecture of the Testbed supports the independent development of new software capabilities by structural analysis researchers and numerical methods developers via the SuperCLIP facility. This facility and the supporting architectural utilities allow the macroprocessor modules and independent programs to access command language symbols and data library contents.

COMPUTER ENVIRONMENT

The CSM Testbed was originally developed on a VAX 11/785 computer using the VMS operating system. In order to address the new computer architectures, it became necessary to migrate to the UNIX environment.

The Testbed relies heavily on the available UNIX tools to provide a common developer interface across the distributed environment. In addition to the UNIX tools, system independent precompilers that support conditional compilation and text insertion complete the requirements for maintaining the software for the distributed environment.

Machines

The Testbed is currently operational on the following types of computers: VAX/VMS, MicroVAX^{TM}/ULTRIX^{TM}, SUN^{TM}/UNIX, FLEX/32^{TM}/UNIX, and CRAY-2/UNICOS. This wide range of computer capabilities create a development environment that makes maximum use of computers at all levels of capability. It is possible to begin an application task at a small single user workstation, develop and test new algorithms using small test cases on a minicomputer, and apply those algorithms on large complex structures using the resources of a supercomputer, all under one application environment, all without modifying any of the Fortran code from that developed on the original workstation.

A single application task often spans the entire range of computers. Model development usually occurs on a workstation, and the final analysis is usually performed on a supercomputer. Although the data libraries are not readable in binary form among the nonhomogeneous computers, the Testbed has commands which allow the user to format data into text files on one computer, and after transferring the text files to the target computer, to restore the data on the target computer for further Testbed processing. These commands are typically used to transfer analysis results from the CRAY-2 to a local MicroVAX workstation for graphics postprocessing.

Distributed Environment

The CSM computers at NASA Langley are linked via the Langley local area network to the NASNET computer network as well as many other government and university computer systems. This network gives researchers the capability of developing and testing new methods in several different computer environments, selecting the machine characteristics which are appropriate for the type of analysis to be performed.

All computers available to the NASA Langley CSM researchers are available through the Internet TCP/IP communication protocol. Individual computers are linked via ethernet within buildings at Langley Research Center. Gateways are provided between buildings by a Pronet-10^{TM} token passing ring. A one megabit/second communication link using a Vitalink^{TM} bridge over a terrestrial, T1, circuit connects Langley Research Center to the Ames Research Center and provides the backbone for the Langley-Ames NASNET connection to the NAS CRAY-2. The resulting network has provided an effective interactive capability as well as high speed file transfer for CSM researchers. Supercomputer resources are provided directly to the individual workstation. The high speed of the long distance communication link gives almost transparent interactive use as well as access to files. Even the large data library files created by the Testbed are easily accessible under this network for transmission to graphics workstations for postprocessing.

THE PROGRAM CODE

Master Source Code

The program code for all target versions of the Testbed is maintained in single copies of the source files, in a format called Assembled Master Source (AMS) form. Embedded preprocessing commands allow selective conditional precompilation by machine-independent utility programs. An example of this form is given in Appendix B, which contains a listing of the main program for the Testbed macroprocessor in the AMS format. Procedures for extracting specific target versions of the compiler source code have been developed for all the systems on which the Testbed has been installed. All source code files and procedures are maintained with the Revision Control System on a MicroVAX computer using the ULTRIX operating system.

The architecture code is made up of approximately 650 modules with about 83000 lines in source code and include files. The application code is made up of approximately 1300 modules with about 95000 lines in source code and include files. Distribution of the code in the UNIX environment is accomplished by packaging the source code, makefiles, and scripts in a single file using the tape archive utility (tar); this distribution file occupies approximately 8 megabytes on disk.

Machine Independent Tools

Two utility programs, MAX and INCLUDE, which operate on the master source files were originally developed by Carlos Felippa at Lockheed Palo Alto Research Laboratory (ref. 12). The MAX utility allows distribution of source code for selected target compilers, computers, and operating systems from a Master Source file which supports the targets. The INCLUDE utility allows text insertion from files named in the source code, similar to the "include" facility of VAX Fortran, in a machine independent manner. Both of these utilities have been modified from the original VAX versions to execute in a UNIX environment and under the UNICOS operating system on the CRAY-2 computer. The main program of the Testbed macroprocessor is presented in Appendix B along with the two specific include files required for compilation. The processor names that are known to the macroprocessor are established in the file named procs.inc. The correspondence between the processor name and the subroutine called when the name is encountered in a command is established in the file named subcalls.inc. These files are inserted into the Fortran source prior to MAX precompilation. Maintaining the processor-specific information in the separate include files allows the macroprocessor to be customized for the application by changing only the two included files and recompiling the main program.

Languages

Fortran is the primary language for the Testbed source code. The Testbed architecture modules are written in Fortran-77, with extensive use of character variables, and a small amount of assembler and C code in the low-level I/O modules for the data management functions. The application modules are written in both Fortran-66 and Fortran-77, the combination of which has presented some limitations to naming conventions for data base entities and processors because of the old code used as a core analysis capability. However, the flexibility of the supporting NICE modules is maintained for use by new application modules.

<u>Procedures for Building</u>

Procedures and scripts for preprocessing, compiling, and linking the Testbed have been developed for the target systems. The scripts on all the UNIX-type systems are almost identical, with differences in the keys used for MAX precompilation, the name and options for the Fortran compiler to be used, and the name and options for the loader program to be used. A set of "makefiles" is used, with a top-level "make" invoking lower-level "makes" to create the required object files and libraries in the correct order. The procedures for building the code under a VMS^{TM} operating system are similar in organization but are implemented in the DCL command language without all the power and flexibility of the UNIX make utility.

VERIFICATION PROCEDURES

In order to verify the correct installation of the Testbed code, programs which test the operation of the programmer interface with the command language interpreter and with the data manager are built on the target computer and executed after the object libraries for that software have been created. Correct results (manually verified at present) from executing these programs verify the installation of the command language and data manager. Once this successful installation has been established, the macroprocessor is built and the scripts for the demonstration problems may be executed. These scripts are written in CLAMP, which is portable across all the different computer systems where the Testbed has been installed. System dependent commands such as those for deleting files, redirecting input and output, and invoking execution of the Testbed are the only differences in the text of the demonstration problem scripts. These scripts also serve the purpose of providing a variety of examples of Testbed usage for new users.

CRAY-2 IMPLEMENTATION

Installation of the Testbed on the NAS CRAY-2 computer was accomplished over a period of about one month in 1987 shortly after the computer was made available to NASA Langley users. The Testbed code was the largest software system to be ported to the NAS computer, and consequently many problems which had not been experienced by other users had to be diagnosed and overcome.

The first step of the installation, which had to be performed before any compilation could be done, was to build the MAX and INCLUDE utilities under UNICOS. This required writing an interface between the Fortran program and the C language argc and getarg functions not provided in the Fortran libraries. Next, compilation of the NICE software was accomplished and object libraries created. The test programs for the NICE software were built and executed successfully. Then the macroprocessor, application modules, and utility routines were compiled. Finally, the linking of the executable file was performed.

Compilation Problems

Because the Testbed Fortran code uses character variables heavily, the CFT77^{TM} compiler had to be used for compilation. Most problems encountered with this compiler were related to its handling of character variables and formatting screen and printed output and were not encountered until execution time. Most of these were resolved by inserting code blocks for the CRAY/UNICOS version into the master source files so that the modifications could be carried along into future versions of the code. The porting of the Testbed to the CRAY-2 was accomplished using a very early version of CFT77 under UNICOS. Although several compiler errors were discovered with that compiler, no errors that could not be easily avoided were uncovered. The compiler errors that were discovered have been fixed in subsequent releases of the CFT77 compiler.

Fortran/C Interface

One problem related to CFT77 character handling which had to be resolved twice was the difference in data structures for CFT77 character arguments and C compiler character string arguments. This problem arises where the Fortran code for the data management functions calls low-level C language I/O functions. The CFT77 compiler does not conform to the same standard as the Fortran compilers on other UNIX-type systems. To overcome the problem, a C structure was defined in the C functions to correspond to the CFT77 character argument; upon entry to the C function, a transformation was performed from the argument structure to a C character string. When version 3.0 of UNICOS was installed with a new CFT77 compiler, the CFT77 character variable structure was changed without documentation, so the C functions had to be modified to accommodate the new structure after the difference was discovered. The definition of the Fortran and C character pointer structures under UNICOS 3 are:

```
typedef struct { ushort offset:  3;          /* string offset in bytes */
                 ushort filer1:  3;          /* length 'bits' count    */
                 ushort length: 23;          /* string length in bytes */
                 ushort filer2:  3;          /* offset 'bits' count    */
                 ushort addres: 32;          /* string address         */
               } chptrf;                     /* CHar PoinTeR Fortran   */
typedef struct { ushort offset:  3;          /* string offset in bytes */
                 ushort filer3: 29;          /* unused space           */
                 ushort addres: 32;          /* string address         */
               } chptrc;                     /* CHar PoinTeR C-lang.   */
```

The called C function assigns the offset and address fields of the input Fortran pointer to the respective fields of the C pointer before moving the characters to a local character array. The use of these structures is illustrated in the block I/O routines in Appendix A.

Loader Problems

The initial installation procedures used the LD loader for linking the executable file. When the optimization options for the CFT77 compiler had been used in compilation, all subroutine argument addresses and some temporary variables were defined in local memory by the compiler. The LD loader concatenates the local memory segments for all modules, so attempting to link all of the application modules and libraries in the macroprocessor resulted in overflow of local memory (40000_8 words) and failure of the load. The LMSTAK utility to enable overlaying local memory segments was used, but the resulting program would not execute. In order to check out the operation of the software before resolving the local memory overflow problem, all the code was recompiled without optimization, linked successfully, and tested.

Later, following the suggestion by the NAS CRAY analysts, the segmentation loader (SEGLDR^{TM}) was used. This loader performs the local memory overlay correctly, so the optimized object code could be used. No execution errors were encountered as a result of using the optimizing compiler. Performance was improved by a factor of 3 in CPU usage with the optimized code for most of the demonstration problems executed. However, vectorization is not used efficiently in this version of the code because of the short vector lengths actually used (≤ 6 in a critical area). Much greater improvements should be gained by tailoring the matrix operations in the code to take advantage of vectorization.

Installation of a new CFT77 compiler with options to enable the user to control the allocation of local memory has since eliminated the requirement to use SEGLDR for the Testbed to overlay local memory.

Optimization

In order to identify the most promising areas for performance improvement, two utilities were used. First, the FLOW utility was used, after recompilation of the code with the CFT77 flowtrace (-ef) option. The resulting executable file was used to run several demonstration problems performing different types of analysis functions. The FLOW utility analyzed the output files and identified the modules which were using most of the CPU time for the executions. A calling tree diagram was also obtained in the FLOW output, which was helpful in analyzing the execution path of the program.

After identifying the biggest CPU users, the Fortran source code for those modules was sent to an IRIS™ workstation on which the FORGE™ software was installed. FORGE was used to insert timing function calls into the modules which were then sent back to the CRAY, compiled and linked into the executable. The demonstration problems were run again and very detailed analyses of the execution of the modules of interest were obtained. These analyses led to replacement of some code with UNICOS library function calls and some other minor revisions. This work resulted in an improvement of about 12% in the performance of the affected analyses.

This installation of the Testbed on the CRAY-2 is allowing researchers to analyze much larger problems in a reasonable turnaround time than has been possible with the minicomputer installations previously available.

STRUCTURAL ANALYSIS METHODS DEVELOPMENT EXPERIENCES

Workstation development of new analysis code and procedures

Researchers at NASA Langley have been using a distributed computer environment to develop new analysis modules and procedures, with each researcher working in the Testbed environment on a local computer or workstation of his preference. Procedures, scripts and makefiles similar to those for maintaining the complete Testbed system are available to the Testbed developers for building external processors or customizing macroprocessors.

The researcher has to be concerned only with the code for his new module, calling utility subroutines from Testbed libraries, where applicable, and using the interface routines for command input and data management functions. There are minimal requirements prescribed for initialization and termination to ensure compatibility with installed processors. Where a new module must interact with other Testbed modules via the data base, it must conform to the data structures defined by the existing modules. The new module must have a name different from any analysis module installed in the Testbed if it is to be executed within a CLAMP procedure along with Testbed modules. Typically, the module is tested on the local workstation or minicomputer to verify its operation before it is sent to the CRAY-2 for further testing.

Once the new module has been initially tested on a local computer, the source code and/or procedures are transferred to the NAS CRAY-2 system via a network, and an executable file is built on the CRAY using the same procedures as on the local computer. CLAMP scripts for verifying the processor operation on the local computer are also portable from the local computer to the CRAY-2.

CRAY-2 UNICOS Environment

A shell script and makefile used for building an external processor on the CRAY-2 computer are shown in Figures 2 and 3. The script refers to an environment variable CSM_ROOT which contains the name of the root directory for the Testbed software files. This variable is passed to the makefile as a macro variable to be used for defining the names of utility object files and library files to be linked with the new module. The makefile uses the MAX and INCLUDE utilities and requires that the user have his PATH environment variable defined so that those files are accessible.

A login script is provided for Testbed users to execute to define their environments for compatibility with these scripts. The user should determine the pathname for the root directory of the Testbed files on each of the computers where the Testbed is to be used. To execute the Testbed login script on a particular computer at login time, the user should insert the following commands in his ".login" file:

```
setenv CSM_ROOT "system_dependent_path"
source $CSM_ROOT/login
```

To precompile, compile, and link a new module into an executable external processor, the following command is used:

```
bldextp module_name [object_file_names]
```

where module_name is the root name of the module source file, which resides in the current directory with extension ".ams"; the optional argument, object_file_names, is a list of object files to be linked with the module. The script automatically links in the Testbed utilities, so these do not have to be included in the list.

The command to execute the Testbed macroprocessor is:

```
testbed
```

The user's PATH environment variable is defined in the login script so that the directory in which the Testbed executable resides is searched by the shell when the above command is entered.

An example shell script for executing the Testbed in conjunction with an external processor is shown in Figure 4.

FUTURE DIRECTIONS FOR THE CSM TESTBED

The future directions of the CSM Testbed will be tied to developments in several other areas, particularly the evolving computer hardware industry. The changes in computer hardware will, out of necessity, result in changes to operating systems and systems software in order to take advantage of the changes in hardware. New approaches in applying numerical analysis will result from changes in computer hardware and software. This evolving technology provides more and faster computer architectures but only at the cost of software compatibility and complexity.

The CSM Testbed is being extended to exploit the multiple instruction multiple data (MIMD) computers that are becoming generally available. To support analysis on MIMD computers, the command language is being rewritten and advanced numerical algorithms are being developed

Command Language Enhancements

In order to provide a better Testbed environment, enhancements to the command language are being developed. The current command language capability was developed over the course of a decade (ref. 13). The present version of the command language interface program (CLIP) contains 129 subroutines and 18,000 lines of source code.

Enhancements to CLIP are underway to include the implementation of a table driven parser and lexical analyzer. The UNIX utilities LEX and YACC will be used to implement an easily extendable language. This language will be primarily the CLAMP language implemented by Felippa in the NICE computer environment with modifications to remove context sensitive constructs from the language. Care is being taken to retain all the problem solving capability proven effective over the last decade while adding generality. As a side benefit we expect the resulting interpreter to be more efficient and maintainable in addition to providing the required extendability. This extendability will be tested by the addition of language directives to control processor/task allocation and synchronization at a high level through CLAMP directives. The resulting capability will provide a convenient research environment for the structural analyst to investigate parallelism without relying on computer dependent coding.

Advanced Numerical Algorithms

Numerical analysts in the CSM activity are developing many new algorithms designed to take advantage of the vector processing capability offered by many modern computers. In the past, the sparse nature of the matrices that dominate structural analysis computations has made vector processors of limited use. Now, however, in addition to numerical algorithms for vector computers, CSM researchers are developing algorithms for MIMD computers. Recent research on algorithms for vector and MIMD computers are described in references 14, 15, and 16. Work will continue on the development of numerical algorithms that will take advantage of both the vector capabilities and the MIMD capabilities of future computers.

CONCLUDING REMARKS

The CSM Testbed is a useful and powerful development environment for developing structural analysis and computational methods. The Testbed development environment provides the mechanism to allow researchers concentrating on different parts of the structural analysis problem to communicate on solutions to problems that directly relate to current NASA needs. The transfer of technology among researchers in computer science, numerical analysis, and structural engineering can now be accomplished more effectively than was previously possible.

The CRAY-2 provides an extremely powerful top-end capability for performing structural analysis applications in a networked distributed environment. It is possible for the same Testbed applications runstream to be used on computers ranging from a workstation running UNIX through the CRAY-2 supercomputer. A runstream may now be checked out on a workstation for a small model prior to performing fullscale calculations on the CRAY-2.

Since the CSM Testbed was operational in a UNIX environment prior to converting to the CRAY-2, the implemention under UNICOS was accomplished without significant problems. The Testbed program was made operational under a pre-release CFT77 compiler. Although several compiler errors were found, corrections were possible with the help of the Cray analysts.

Planned development of the CSM Testbed on supercomputers will involve extensions that will allow researchers to develop combined vector/MIMD applications methods in an integrated environment. The integrated environment is characterized by a common operating system, common file system, and usually a common administrative system.

ACKNOWLEDGEMENTS

Prior experience of Dr. Frank Weiler of the Lockheed Palo Alto Research Laboratory with the UNICOS compilers during installation of the STAGS-C1 structural analysis code on the CRAY-2 was extremely valuable to the success of the Testbed installation. Assistance and advice given by the NAS consultants is gratefully acknowledged.

REFERENCES

1. Knight, Norman F., Jr.; and Stroud, W. Jefferson: *Computational Structural Mechanics: A New Activity at the NASA Langley Research Center*. NASA TM-87612, September 1985.
2. McLean, Donald M.: *MSC/NASTRAN Programmer's Manual, MSC/NASTRAN Version 63*. MSR-50, pp. 1.1-4, October 1983.
3. Knight, Norman F., Jr.; Gillian, Ronnie E.; and Nemeth, Michael P.: *Preliminary 2-D Shell Analysis of the Space Shuttle Solid Rocket Boosters*. NASA TM-100515, November 1987.
4. Felippa, Carlos A.: *Architecture of a Distributed Analysis Network for Computational Mechanics*. Computers and Structures, Vol. 13, 1981, pp. 405-413.
5. Felippa, Carlos A.: *The Computational Structural Mechanics Testbed Architecture: Volume 2 - Directives*. NASA CR-178385, 1988.
6. Felippa, Carlos A.: *The Computational Structural Mechanics Testbed Architecture: Volume 1 - The Language*. NASA CR-178384, 1988.
7. Wright, Mary A.; Regelbrugge, Marc E.; and Felippa, Carlos A.: *The Computational Structural Mechanics Testbed Architecture: Volume 4 - The Global-Database Manager GAL-DBM*. NASA CR-178387, 1988.
8. Hurst, P. W.; and Pratt, T. W.: *Executive Control Systems in the Engineering Design Environment*. AIAA Paper No. 85-0619, 1985.
9. Felippa, Carlos A.: *Fortran-77 Simulation of Word-Addressable Files*. Advanced Engineering Software, Vol. 4, Number 4, 1982, pp. 156-162.
10. Lotts, C. G.; Greene, W. H.; McCleary, S. L.; Knight, N. F., Jr.; Paulson, S. S.; and Gillian, R. E.: *Introduction to the Computational Structural Mechanics Testbed*. NASA TM-89096, 1987.
11. Knight, N. F.; McCleary, S. L.; and Macy, S. C.; and Amminpour, Mohammad A.: *Large-Scale Structural Analysis: The Structural Analyst, the CSM Testbed, and the NAS System*. NASA TM-100643, 1988.
12. Felippa, Carlos A.: *MAX and Friends*. NASA CR-178383, 1988.
13. Felippa, Carlos A.: *A Command Reader for Interactive Programming*. Engineering Computations, Vol. 2, Number 3, Sept. 1985, pp. 203-237.
14. George, J. A.; and Liu, J. W. H.: *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, N. J., 1981.
15. Poole, E. L.; and Overman, A. L.: *The Solution of Linear Systems of Equations with a Structural Analysis Code on the NAS CRAY-2*. NASA CR-4159, 1988.
16. Storaasli, O. O.; Poole, E. L.; Ortega, J.; Cleary, A.; and Vaughan, C.: *Solution of Structural Analysis Problems on a Parallel Computer*. AIAA Paper Number 88-2287, 1988.

TRADEMARKS

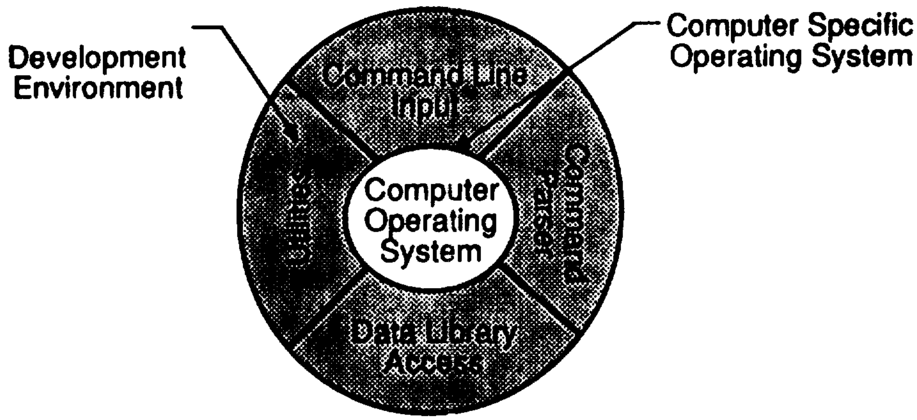

Figure 1. CSM Testbed Organization

```
#! /bin/sh
#
# Shell script for invoking make to build an external processor executable;
# Environment variable CSM_ROOT must be defined prior to invoking this script.
#------------------------------------------------------------------------
# Usage: bldextp processor_name [other_objects]
# where: processor_name is the name of the executable file to be built;
#                        (the source code for the processor must be in
#                        a file of the same root name with extension .ams)
#        other_objects is an optional list of full pathnames of object files
#                        to be built and linked with the processor object,
#                        Testbed utilities and NICE libraries
#------------------------------------------------------------------------
case $# in
0)    echo 'Usage: bldextp processor_name [other_objects]' 1>&2 ; exit 2;;
2)    EXTP=$1; shift
esac
#------------------------------------------------------------------------
# Check to make sure shell variables are defined
#------------------------------------------------------------------------
case "$CSM_ROOT" in
"")   echo 'CSM environment variables must be set before invoking bldextp.' 1>&2
      exit 2
esac
#------------------------------------------------------------------------
# make other object files first
#------------------------------------------------------------------------
for i in $*
do
  echo Making $i
  FILE='basename $i .o'
  DIR='dirname $i'
  ( case "$DIR" in
    "") ;;
     *)  cd $DIR
    esac
    make -f $CSM_ROOT/sam/tbo.mk $FILE.o CSM_ROOT=$CSM_ROOT \
          MAXKEYS="NICE EXTP TEK" )
  case $? in
  0) ;;
  *) exit 2
  esac
done
#------------------------------------------------------------------------
# make external processor object and executable
#------------------------------------------------------------------------
echo Making $EXTP
FILE='basename $EXTP'
DIR='dirname $EXTP'
FILE='basename $FILE .o'
( case "$DIR" in
  "") ;;
   *)  cd $DIR
  esac
 make -f $CSM_ROOT/sam/tbo.mk EXE=$FILE CSM_ROOT=$CSM_ROOT OBJS="$*" \
      MAXKEYS="NICE EXTP TEK" )
```

Figure 2. Shell script for building a Testbed external processor

```
# tbo.mk
#
# Makefile for building a Testbed object or executable file
#
#   Build an object file for a Testbed module from an AMS file.
#   Link the object files with Testbed utility object files and
#   NICE libraries.
#
#-----------------------------------------------------------------------------
# Macros (Make macros which may be overridden on the make command)
#-----------------------------------------------------------------------------
# CSM_ROOT must be defined on the make command
CSM_MOD = $(CSM_ROOT)/sam/mod
CSM_UTL = $(CSM_ROOT)/sam/utl
EXE = testbed
FC = cft77
FFLAGS = -a static -ecrsx
INCDIR = .
LD = segldr
LFLAGS =
MAXKEYS = NICE TEK
NICELIBS = $(NLB)/clp861b.a \
        $(NLB)/gal861b.a \
        $(NLB)/dmg861b.a \
        $(NLB)/utl861b.a \
        $(NLB)/bio861b.a
NLB = $(CSM_ROOT)/nice/lib
OBJS =
UTILS = $(CSM_MOD)/gsutil.o \
        $(CSM_MOD)/nsutil.o \
        $(CSM_MOD)/nsparlib1.o \
        $(CSM_MOD)/nsparlib2.o \
        $(CSM_MOD)/nsparlib3.o \
        $(CSM_UTL)/plot10.a
#--------------------------------------------------------------------------
# Suffix Rules
#--------------------------------------------------------------------------
.SUFFIXES:
.SUFFIXES: .ams .o
# Transform .ams file to .o file; use include and max utilities, then compile
.ams.o:
        include -i $*.ams -o $*.tmp -d $(INCDIR)
        - rm $*.f
        max /wc/uc/for/sic/ti/mach=unix -i $*.tmp -o $*.f $(MAXKEYS)
        - rm $*.tmp
        $(FC) $(FFLAGS) $*.f
#       rm $*.f
#-----------------------------------------------------------------------------
# Targets
#-----------------------------------------------------------------------------
# Executable depends on an object file with the same root name ;
# link with named objects, utilities, and NICE libraries
$(EXE): $(EXE).o
        $(LD) $(LFLAGS) -o $@ $@.o $(OBJS) $(UTILS) $(NICELIBS)
```

Figure 3. Makefile for building a Testbed external processor

```
time testbed << \eof
*SET ECHO OFF
*open 1 cube.101
*PROCEDURE CUBE
*DEF NN == 7
*DEF/g LL == 22.222222
*DEF NNM1 =< <NN> - 1>
*DEF NNNN =< <NN>*<NN> >
*DEF JNT =< <NNNN>*<NN> >
[XQT TAB
 online=0
 START <JNT> 4,5,6
[XQT AUS
 online=0
 TABLE(NI=31,NJ=1): PROP BTAB 2 21
 J=1
 .101>
 .1E-6>
 -.3E-7 .1E-6>
 -.3E-7 -.3E-7 .1E-6>
 0.0 0.0 0.0 .26E-6>
 0.0 0.0 0.0 0.0 .26E-6>
 0.0 0.0 0.0 0.0 0.0 .26E-6>
 0.0 0.0 0.0>
 1.0 1.0 1.0 1.0 1.0 1.0
[XQT TAB
 online=0
 JLOC
*DEF N = <NN>
*DEF/i JCNT = 1
*DEF/g Z = 0.0
*DEF/g DELZ =< <LL>/<NNM1> >
*DO $I=1,<N>
 <JCNT> 0. 0. <Z> <LL> 0.0 <Z> <NN> 1  <NN>
  <NN> 0. <LL> <Z>   <LL> <LL> <Z>
*DEF/g Z =< <Z> + <DELZ> >
*DEF/i JCNT =< <NNNN> + <JCNT> >
*ENDDO
 MATC : 1 10.+6 .3 .101
 CON 1 : ZERO 1,2,3: 1 : <NN> : <NNNN>
 *DEF N =< <NNNN> - <NN> + 1 >
 <N>
[XQT ELD
 online=0
 S81
 1 <NNM1> <NNM1> <NNM1>  1  <NN> <NNNN> 0 1
[xqt pfmx  . Execute external processor pfmx (experimental version of PFM)
     reset method=5,maxcon=8,nalg=0
[XQT TOPO
     reset maxsub=40000, LRAM=12288
     stop
*sho macros
[xqt dcu
     toc 1
*END
*CALL CUBE
[XQT EXIT
\eof
```

Figure 4. Example script executing Testbed with external processor

Appendix A. Testbed CRAY-2 Block I/O Routines

IOXCLO - Close a file

```
/*  IOXCLO:  close (/delete) a file from randon i/o, given it's descriptor
 *  Modified for UNICOS 3.0 cft77 character arguments
 *
 *  this routine is called in fortran (f77) via:
 *
 *       call  IOXCLO (fd, opt, path, size, bkez, blks, msg )
 *      -------------------------------------------------
 *
 *  input arguements:
 *
 *    fd     =  file descriptor for file (for closing)
 *    opt    =  1 character close option flag, where
 *              opt = ' ' for normal close
 *                    'd' for close/delete option
 *    path   =  complete 'path' name of file (for delete opt)
 *
 *  output arguements:
 *
 *    size   =  size of file (in bytes)
 *    bksz   = optimum block size of file
 *    blks   = number of blocks in file
 *    msg    = error return message (blank if no error)
 *
 */

#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/file.h>
#include   <sys/stat.h>

typedef struct { ushort offset:  3;
                 ushort filer1:  3;
                 ushort length: 23;
                 ushort filer2:  3;
                 ushort addres: 32;} chptrf;

typedef struct { ushort offset:  3;
                 ushort filer3: 29;
                 ushort addres: 32;} chptrc;

/*********************
 *  entry - IOXCLO  *
 *********************/

IOXCLO(fd, opt, path, size, bksz, blks, msg)

chptrf     path,  opt, msg ;
long int   *fd,  *size, *bksz, *blks ;
```

```
{

     extern int   errno;
     extern int   BLKSIZ;

     chptrc       cpath, copt;
     int          lpath, lopt, lmsg ;
     char         *op, *pp, fnam[256];

     int          i;
     struct stat  stbuf;

     errno = 0;                        /* clear the error code no.  */

     /* create local C-pointers form FORTRAN-pointers */

     cpath.offset =  path.offset;
     cpath.addres =  path.addres;
     lpath        =  path.length;

     copt.offset  =  opt.offset;
     copt.addres  =  opt.addres;
     lopt         =  opt.length;

     lmsg         =  msg.length;

     pp           =  cpath;
     op           =  copt;

     /* first, extract the file's stats for return */

     if (fstat(*fd, &stbuf) == -1)     /* error in status request ? */
       {
       *size = 0;                      /* clear file statistics  */
       *bksz = 0;
       *blks = 0;
       IOXERR_(errno, msg, lmsg);            /* extract error number   */
       }
     else                              /* valid status info      */
       {
       *size = stbuf.st_size;          /* set 'size' = file size (char) */
/*
 *       *bksz = stbuf.st_blksize;
 *       *blks = stbuf.st_blocks;
 */
       IOXERR_(NULL, msg, lmsg );            /* clear error number              */
       }

     /* second, close the file pointed to by 'fd' */

     if (close(*fd) == -1)             /* error on close request ? */
       {
```

```
      IOXERR_(errno, msg, lmsg);          /* extract error number   */
      return;
      }

    /* last, check option and delete file if request is on */

    if (*op == 'd')                    /* delete option on ? */
      {

      /* transfer FORTRAN character string (path) to local array fnam[] */

      i   =  0;
      while ((*pp != '\0') && (*pp != ' ') && (i < lpath))
        {
        fnam[i++] = *(pp++);
        }
      fnam[i] = '\0';

      if (unlink(fnam) == -1)          /* error on delete request ? */
        {
        IOXERR_(errno, msg, lmsg);        /* extract error number    */
        }
      }
    return;
}
```

IOXERR - Return a system error message, given the error no.

```
/*  IOXERR:  return a system error message, given 'erno'
 *  Modified for UNICOS 3.0 cft77 character arguments
 *
 *      IOXERR_(erno, msg, lmsg )
 *
 *  input arguements:
 *
 *    erno     =  system error number
 *                if erno =         0,  NULL error, ierr = 0
 *                   erno <         0,  undefined,  ierr = -1
 *                   erno >  sys_nerr,  undefined,  ierr = -1
 *    lmsg      = length of message string 'msg'
 *
 *  output arguements:
 *
 *    msg       =  error message (blank if 'erno' == NULL )
 *
 */

#include   <stdio.h>
#include   <errno.h>

IOXERR_(erno, msg, lmsg )

char       *msg;
long int   erno,  lmsg;

{
  extern int   sys_nerr;        /* largest error no. for which system  */
                                /* system tables has a defined message */
  extern char *sys_errlist[];   /* table of system error messages      */

    strncpy ( msg, " ", lmsg );
    if (erno == NULL)           /* NULL error message returned */
      {
      return;
      }

    if (erno>0 && erno<sys_nerr)
      {
      strncpy(msg, sys_errlist[erno], lmsg );
      printf (" IOXERR: ierr = %4d  (%s)", errno, sys_errlist[erno]);
      }
    else
      {
      strncpy(msg, "ERROR: unknown error value", lmsg);
      printf (" IOXERR: ierr = %4d  (unkown error value ?)", errno);
      }
  return;
}
```

IOXLOC - Extract the current position within a file

```
/*  IOXLOC:  extract the current position within a file
 *  Modified for UNICOS 3.0 cft77 character arguments
 *
 *  this routine is called in fortran (f77) via:
 *
 *       call  IOXLOC (fd, size, bksz, blks, pos, msg)
 *      ------------------------------------
 *
 *  input arguements:
 *
 *    fd    =  file descriptor for file
 *
 *  output arguements:
 *
 *    size  =  size of file (in bytes)
 *    bksz  =  optimum block size of file
 *    blks  =  number of blocks in file
 *    pos   =  position within file returned by lseek(2)
 *    msg   =  error return message (blank if no error)
 *
 */

#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/file.h>
#include   <sys/stat.h>

typedef struct { ushort offset:  3;
                 ushort filer1:  3;
                 ushort length: 23;
                 ushort filer2:  3;
                 ushort addres: 32;} chptrf;

/*
 *    the following flags represent file positioning
 *    parameters used by lseek(---)
 *
 */
#define    L_SET     0      /* absolute offset (from BOF) */
#define    L_INCR    1      /* relative to current offset */
#define    L_XTND    2      /* relative to end of file    */

IOXLOC(fd, size, bksz, blks, pos, msg )

chptrf     msg;
long int   *fd, *size, *bksz, *blks, *pos;

{

     int lmsg;
     extern int   errno;
```

```
    extern int   BLKSIZ;
    struct stat  stbuf;

    lmsg = msg.length;

    *pos  =  lseek (*fd, OL, L_INCR);   /* extract position within file */

    if (fstat(*fd, &stbuf) == -1)      /* error in status request ? */
      {
      *size = 0;                       /* clear file statistics  */
      *bksz = 0;
      *blks = 0;
      IOXERR_(errno, msg, lmsg);            /* extract error number   */
      }
    else                               /* valid status info      */
      {
      *size = stbuf.st_size;           /* set 'size' = file size (char) */
/*
 *     *bksz = stbuf.st_blksize;
 *     *blks = stbuf.st_blocks;
 */
      IOXERR_(NULL, msg, lmsg );            /* clear error number            */
      }
    return;
}
```

IOXOPN - Open a file for random I/O

```
/*  IOXOPE:  open a file for randon i/o, given it's path name
 *  Modified for UNICOS 3.0 cft77 character arguments
 *
 *  this routine is called in fortran (f77) via:
 *
 *       call  IOXOPE (path, opt, fd, size, bksz, lbks, msg )
 *      ------------------------------------------
 *
 *  input arguements:
 *
 *    path  =  complete 'path' name of file
 *    opt   =  2 character open option flags, where
 *             opt[0] = 'r' for 'read_only'
 *                      'w' for 'write_append'
 *                      ' ' for both 'read_write'
 *             opt[1] = 'o' for  'existing'  file open
 *                      'n' for 'create_new' file open
 *                      's' for 'scratch'    file open
 *                      ' ' for 'create_new' even if file
 *                          already exists (truncate old)
 *
 *  output arguements:
 *
 *    fd    =  file descriptor for open file
 *    size  =  size of file (in bytes)
 *    bksz  =  optimum block size of file
 *    blks  =  number of blocks in file
 *    msg   =  error return message (blank if no error)
 *
 */

#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/file.h>
#include   <sys/stat.h>
#include   <errno.h>

#include   <fcntl.h>
/*
 *    the following flags represent file
 *    accessing  modes used by open(---)
 *
 */
#define    R_OK     04       /* read permission             */
#define    W_OK     02       /* write permission            */
#define    X_OK     01       /* execute/search permission   */
#define    F_OK     00       /* file existence              */

int   BLKSIZ = 4096;          /* preset block/buffer size
                                 in bytes (= 512 words)     */

```

```
typedef struct { ushort offset:  3;            /* string offset in bytes */
                 ushort filer1:  3;            /* length 'bits' count    */
                 ushort length: 23;            /* string length in bytes */
                 ushort filer2:  3;            /* offset 'bits' count    */
                 ushort addres: 32;            /* string address         */
               } chptrf;                       /* CHar PoinTeR Fortran   */
                                               /* ^^   ^    ^ ^ ^        */

typedef struct { ushort offset:  3;            /* string offset in bytes */
                 ushort filer3: 29;            /* unused space           */
                 ushort addres: 32;            /* string address         */
               } chptrc;                       /* CHar PoinTeR C-lang.   */
                                               /* ^^   ^    ^ ^ ^        */
/**********************
 *  entry  -  IOXOPE  *
 **********************/

IOXOPE(path, opt, fd, size, bksz, blks, msg )


chptrf      path,  opt, msg ;
long int    *fd,   *size, *bksz, *blks ;

{

     extern int   errno;
     extern int   BLKSIZ;

     chptrc       cpath, copt;
     int          lpath, lopt, lmsg;
     char         oc, *op, *pp, fnam[256];

     int          i, flags, mode, acc;
     struct stat  stbuf;

     /* create local C-pointers form FORTRAN-pointers */

     cpath.offset =  path.offset;
     cpath.addres =  path.addres;
     lpath        =  path.length;

     copt.offset  =  opt.offset;
     copt.addres  =  opt.addres;
     lopt         =  opt.length;

     lmsg         =  msg.length;

     pp           =  cpath;
     op           =  copt;

     /* transfer FORTRAN character string (path) to local array fnam[] */

     i   =  0;
```

```
      while ((*pp != '\0') && (*pp != ' ') && (i < lpath))
        {
        fnam[i++] = *(pp++);
        }
      fnam[i] = '\0';

      /* setup open options by checking file status */

      oc    =  'c';                        /* set open/create flag = 'create' */
      acc   =  access(fnam, F_OK);         /* attempt to access the file  */

      if (acc == -1)                       /* file does not exist */
        {                                  /* ================== */
        if (*(op+1)=='o')                  /* user option says 'old' */
          {
          IOXERR_(errno, msg, lmsg );           /* extract error number  */
          return;                          /* return to user */
          }
        }
      else                                 /* file exists */
        {                                  /* =========== */
        if (*(op+1)=='n')                  /* user option says 'new'  */
          oc = 'c';                        /* set open/create flag = 'create' */
        else                               /* user option says 'old/both' */
          oc = 'o';                        /* set open/create flag = 'open' */
        }

      mode  =  0644;                       /* set protection = (rw_,r__,r__) */

      flags =  0;                          /* set open 'flags' using (opt)   */

      if (*(op) == 'r')     flags = flags | O_RDONLY;
      if (*(op) == 'w')     flags = flags | O_WRONLY;
      if (*(op) == ' ')     flags = flags | O_RDWR;

/*    if (*(op+1) == 'o')   flags = flags | O_EXCL;    */
      if (*(op+1) == 'n')   flags = flags | O_CREAT  | O_TRUNC;
      if (*(op+1) == 's')   flags = flags | O_CREAT  | O_TRUNC;


      *size =  0;                          /* set defaults for file size */
      *bksz =  0;
      *blks =  0;

      *fd = open(fnam, flags, mode);       /* open/create file 'path'    */


      if (*fd == -1)                       /* error on open/create request  */
        {
        IOXERR_(errno, msg, lmsg );             /* extract error number           */
        }
      else                                 /* valid file open, check status */
        {
```

```
        if (fstat(*fd, &stbuf) == -1)   /* error in status request ?     */
          {
          IOXERR_(errno, msg, lmsg );         /* extract error number          */
          }
        else                            /* valid status info             */
          {
          *size = stbuf.st_size;        /* set 'size' = file size (char) */
/*
 *         *bksz = stbuf.st_blksize;
 *         *blks = stbuf.st_blocks;
*/
          IOXERR_(NULL, msg, lmsg );          /* clear error number            */

/* if file has been successfully opened and it is a scratch file,
** then if the file is unlinked at this point (while still open),
** it will be deleted when it is closed.
*/
          if  ((*(op+1) == 's') && (unlink(fnam) != 0))
                  IOXERR_(errno, msg, lmsg) ;

          }
        }
     return;
}
```

IOXRDR - Read n words from file

```
/*  IOXRDR:  read 'n' words from file 'fd', starting at block no. 'blk'
 *  Modified for UNICOS 3.0 cft77 character arguments
 *
 *  this routine is called in fortran (f77) via:
 *
 *       call  IOXRDR (fd, buf, nwds, blk, msg )
 *      ----------------------------------------
 *
 *  input arguements:
 *
 *    fd     =  file descriptor for file
 *    buf    =  pointer to buffer string (char)
 *    nwds   =  number of words (long int) to read in
 *    blk    =  starting 'block' number in file
 *
 *  output arguements:
 *
 *    msg    =  error return message (blank if no error)
 *
 */

#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/file.h>

typedef struct { ushort offset:  3;
                 ushort filer1:  3;
                 ushort length: 23;
                 ushort filer2:  3;
                 ushort addres: 32;} chptrf;
/*
 *    the following flags represent file positioning
 *    parameters used by lseek(---)
 *
 */
#define    L_SET     0      /* absolute offset (from BOF) */
#define    L_INCR    1      /* relative to current offset */
#define    L_XTND    2      /* relative to end of file    */

IOXRDR(fd, buf, nwds, blk, msg )

chptrf      msg;

char        *buf;
long int    *fd,  *nwds, *blk;

{

     extern int  errno;
     extern int  BLKSIZ;
     int         nbuf, pos, ibuf, lmsg;
```

```
    long int    offset;

    lmsg    = msg.length;

    nbuf    =   *nwds * 8;              /* no_bytes = 8 * no_words      */
    offset =  BLKSIZ * (*blk - 1);      /* (byte_wise) offset of 'blk' */

    if (offset >= 0)                    /* position file before read request */
      {
      pos    =  lseek(*fd, offset, L_SET);

      if (pos == -1)                    /* error condition in lseek call  */
        {
        IOXERR_(errno, msg, lmsg );         /* extract error number           */
        return;
        }
      }

    ibuf  =  read(*fd, buf, nbuf);      /* read in 'nbuf' bytes to 'buf' */

    if ((ibuf == -1) || (ibuf != nbuf))     /* error condition on read */
      {
      IOXERR_(errno, msg, lmsg);            /* extract error number */
      }
    else
      {
      IOXERR_(NULL, msg, lmsg );            /* clear error number */
      }
    return;
}
```

IOXWTR - Write n words to a file

```
/*  IOXWTR:  write 'n' words to file 'fd', starting at block no. 'blk'
 *  Modified for UNICOS 3.0 cft77 character arguments
 *
 *  this routine is called in fortran (f77) via:
 *
 *       call  IOXWTR (fd, buf, nwds, blk, msg)
 *      -----------------------------------------
 *
 *  input arguements:
 *
 *    fd    =  file descriptor for file
 *    buf   =  pointer to buffer string (long int)
 *    nwds  =  number of words (long int) to write out
 *    blk   =  starting 'block' number in file
 *
 *  output arguements:
 *
 *    msg   =  error return message (blank if no error)
 *
 */

#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/file.h>

typedef struct { ushort offset:  3;
                 ushort filer1:  3;
                 ushort length: 23;
                 ushort filer2:  3;
                 ushort addres: 32;} chptrf;

/*
 *    the following flags represent file positioning
 *    parameters used by lseek(---)
 *
 */
#define    L_SET     0      /* absolute offset (from BOF) */
#define    L_INCR    1      /* relative to current offset */
#define    L_XTND    2      /* relative to end of file    */

IOXWTR(fd, buf, nwds, blk, msg)

chptrf     msg;
char       *buf;
long int   *fd,  *nwds, *blk;

{

     extern int  errno;
     extern int  BLKSIZ;
     int         nbuf, pos, ibuf, lmsg;
```

```
    long int    offset;

    lmsg    = msg.length;

    nbuf    =  *nwds * 8;               /* no_bytes = 8 * no_words     */
    offset =  BLKSIZ * (*blk - 1);      /* (byte_wise) offset of 'blk' */

    if (offset >= 0)                    /* position file before write request */
      {
      pos = lseek(*fd, offset, L_SET);

      if (pos == -1)                    /* error condition in lseek call ? */
        {
        IOXERR_(errno, msg, lmsg );         /* extract error number              */
        return;
        }
      }

    ibuf  =  write(*fd, buf, nbuf);     /* write in 'nbuf' bytes to 'buf'  */

    if ((ibuf == -1) || (ibuf != nbuf))     /* error condition on write */
      {
      IOXERR_(errno, msg, lmsg );           /* extract error number */

      }
    else
      {
      IOXERR_(NULL, msg, lmsg );            /* clear error number */
      }

    return;
}
```

Appendix B. Testbed Main Program

Testbed Main Program in AMS format

```
C$Header: nicespar.ams,v 1.3.1.1 87/09/21 14:59:54 ns Exp $
C-DECK TESTBED TESTBED FORTRAN
C-BLOCK FORTRAN
      program testbed
c
c main program for CSM Testbed macroprocessor
c
C+--------------------------------------------------------------------+
C+              C O M M O N   &   G L O B A L S                       +
C+--------------------------------------------------------------------+
      include 'CSM_INC:KORCOMA.INC'
      common /iando/ iin, ioutx
      common/nsextp/iextp
C+--------------------------------------------------------------------+
C+                          L O C A L S                               +
C+--------------------------------------------------------------------+
      character*50 verid,vertitl
      character*32 idproc, commnd, cclval
      character*64 procnam, filnam, cclmac
      character*256 image
      character*8 cdt(2)
      logical exis
      integer runmod

C+--------------------------------------------------------------------+
C+                    Installed Processors                            +
C+--------------------------------------------------------------------+
      include 'procs.inc'
C+--------------------------------------------------------------------+
C+                            D A T A                                 +
C+--------------------------------------------------------------------+
      data iextp/1/
      data vertitl/' CSM TESTBED Ver. 1.2 - May 1988'/
C+--------------------------------------------------------------------+
C+                          L O G I C                                 +
C+--------------------------------------------------------------------+
C
C   Initialize common variable to the length of the blank common work array
C
      kort = kszzz                      ! Changed from DATA statement CGL 4/86
C
C   Send empty message to CLIP to force it to boot
C
      call clput(' ')
C
C   Set unit where printed output will be written
C   Look for macrosymbol 'ns_prtunt' first, then CLIP PRT if not defined
C
```

```
      call nsprtu( ioutx, ierr)
C
C=IF VAX
      verid='VAX/VMS'//vertitl
C=ELSEIF UNIX
      verid='UNIX'//vertitl
C=ELSEIF CRAY
      verid='CRAY-2'//vertitl
C=ENDIF
C
C   Get current date and time
C
      call datimc ( 'R', o, cdt )
      write(ioutx, '(/1x,a,10x,a,1x,a/)') verid, cdt(1), cdt(2)
c
      call timrb
      call gmacro(1)
c
  100 continue
      idproc = 'CSM'
      call gmsign(idproc)
  200 continue
c
C   Get next user command
C
      call clnext(' CSM>',' Enter command to execute processor: [XQT  '
     $              //'proc_name', nitems )
      commnd = cclval(1)
      idproc = cclval(2)
C=IF UNIX
      lenc = lenetb(commnd)
      call cc2uc( commnd, commnd, lenc )
      lenp = lenetb(idproc)
      call cc2uc( idproc, idproc, lenp )
C=ENDIF UNIX
c
c   check macrosymbol to see whether or not to initialize blank
c   common array to zero
c
      initcom=iclmac('NS_INITCOM')
c
      if( nitems.gt.1.and.((commnd.eq.'[XQT').or.(commnd.eq.'RUN ')))
     $then
         if(idproc.eq.'EXIT') then
            call clput('*stop')
         else
           if( idproc.ne.' ') then
             do 300 i=1,nproc
             if( idproc .eq. namep(i)) then
              if (initcom .ne. 0) then
                do 290 j=1,kort
  290             a(j)=0.
              endif
```

```
                  iproc = i
                  go to 1000
                endif
  300           continue
C=IF VAX
                procnam='ns$extp:'//idproc//'.exe'
                call stripbl(procnam)
                inquire(file=procnam,exist=exis)
C=ELSEIF UNIX
c
c   Close bulk output file
c
                if ( ioutx.ne.6) close ( unit=ioutx )
c
c   Convert filename to lower case
c
                call cc2lc(idproc,idproc,lenp)
                procnam=idproc
                exis=.true.
C               Let Superclip try to find the file
C=ELSE
                procnam=idproc
                exis=.true.
C               Let Superclip try to find the file
C=ENDIF
                if(exis) then
C
c      run external processor
C
                  call clput('*run '//procnam)
C
C      If we get back here, there was an error;
C      continue if interactive or terminate if batch
C
                  write(ioutx,*) 'Unable to execute ',procnam
                  call fbi(runmod)
  302             if(runmod.eq.0) call endrun ( 'CSM', 302 )
                  go to 100
                else
                  write(IOUTX,310) procnam
  310             format(' Unable to run ',a,'; File not found.')
                endif
C
              else
                write(IOUTX,320)
  320           format(' Error, invalid TESTBED command; image follows:')
                call clglim ( image )
                write(IOUTX,321) image
  321           format(a132)
              endif
            endif
         else
            write(IOUTX,320)
```

```
            call clglim ( image )
            write(IOUTX,321) image
         endif
         go to 100
 1000 continue
c
c execute the appropriate processor
c
         include 'subcalls.inc'
 1500 continue
         go to 100
         end
C=END FORTRAN
```

Include File Containing Processor Abbreviations - (procs.inc)

```
      parameter (nproc =39)
      character*6 namep(nproc)
      data namep/'AUS','DCU','DR','E','EIG','EKS','ELD','EQNF','GSF',
     $           'INV','K','KG','M','PS','PLTA','PSF','SSOL','TAB',
     $           'TOPO','VPRT','VEC','IMP','PAMA','PKMA','PRTE','LAU',
     $           'CSM1','RSEQ','PFM','NTP','SSTA','TAFP','TGEO','TRTA',
     $           'TRTB','TRTG','TAK','TADS','VIEW'/
```

Include File for Macroprocessor Subroutine Calls - (subcalls.inc)

```
      go to(1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1011,1012,
     $      1013,1014,1015,1016,1017,1018,1019,1020,1021,1022,1023,1024,
     $      1025,1026,1027,1028,1029,1030,1031,1032,1033,1034,1035,1036,
     $      1037,1038,1039)
     $ ,iproc
 1001 call AUS
      go to 200
 1002 call DCU
      go to 200
 1003 call DR
      go to 200
 1004 call E
      go to 200
 1005 call EIG
      go to 200
 1006 call EKS
      go to 200
 1007 call ELD
      go to 200
 1008 call EQNF
      go to 200
 1009 call GSF
      go to 200
 1010 call INV
      go to 200
 1011 call K
      go to 200
 1012 call KG
      go to 200
 1013 call M
      go to 200
 1014 call PS
      go to 200
 1015 call PLTA
      go to 200
 1016 call PSF
      go to 200
 1017 call SSOL
      go to 200
 1018 call TAB
      go to 200
 1019 call TOPO
      go to 200
 1020 call VPRT
      go to 200
 1021 call VEC
      go to 200
 1022 call IMP
      go to 200
 1023 call PAMA
      go to 200
```

```
1024 call PKMA
     go to 200
1025 call PRTE
     go to 200
1026 call LAU
     go to 200
1027 call CSM1
     go to 200
1028 call RSEQ
     go to 200
1029 call PFM
     go to 200
1030 call MPMTP
     go to 200
1031 call MPSSTA
     go to 200
1032 call MPTAFP
     go to 200
1033 call MPTGEO
     go to 200
1034 call MPTRTA
     go to 200
1035 call MPTRTB
     go to 200
1036 call MPTRTG
     go to 200
1037 call MPTAK
     go to 200
1038 call MPTADS
     go to 200
1039 call MPVIEW
     go to 200
```

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No. NASA TM-100642	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle The CSM Testbed Software System: A Development Environment for Structural Analysis Methods on the NAS CRAY-2		5. Report Date September 1988
		6. Performing Organization Code
7. Author(s) Ronnie E. Gillian and Christine G. Lotts		8. Performing Organization Report No.
9. Performing Organization Name and Address NASA Langley Research Center Hampton, VA 23665-5225		10. Work Unit No. 505-63-01-10
		11. Contract or Grant No.
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546-0001		13. Type of Report and Period Covered Technical Memorandum
		14. Sponsoring Agency Code

15. Supplementary Notes
Ronnie E. Gillian, Langley Research Center, and Christine G.Lotts, Planning Research Corporation

16. Abstract
The Computational Structural Mechanics(CSM) Activity at Langley Research Center is developing methods for structural analysis on modern computers. To facilitate that research effort, an applications development environment has been constructed to insulate the researcher from the many computer operating systems of a widely distributed computer network. The CSM Testbed development system was ported to the Numerical Aerodynamic Simulator(NAS) Cray-2, at the Ames Research Center, to provide a high end computational capability. This paper describes the implementation experiences, the resulting capability, and the future directions for the Testbed on supercomputers.

17. Key Words (Suggested by Authors(s)) Structural analysis software; Finite element analysis; Finite element software		18. Distribution Statement Unclassified—Unlimited; Subject Category 39	
19. Security Classif.(of this report) Unclassified	20. Security Classif.(of this page) Unclassified	21. No. of Pages 45	22. Price A03

NASA FORM 1626 OCT 86

For sale by the National Technical Information Service, Springfield, Virginia 22161-2171