# CSM Testbed Architecture

by
Philip Underwood
Lockheed Palo Alto Research Laboratory
Computational Mechanics Section

Abstract

This presentation on the CSM Testbed Architecture includes the background for the CSM Testbed Architecture and a description of the three architecture related tasks under the NASA/Langley CSM contract (NAS1-18444): 1) Task 2 — Near-Term Enhancements, 2) Task 5 — Matrix Algebra Methods and Utilities, and 3) Task 6 — Data Management for Parallel Computers. For each task the objectives, subtasks chosen to achieve the objectives, and the accomplishments are presented.

For Task 2, Near-Term Enhancements to the CSM Testbed Architecture, the primary objectives are to quickly correct: 1) inefficiency in the GAL data manager, and 2) deficiency in the CLIP-interpreted command language, CLAMP. The corresponding modifications should preserve upward-compatibility to a reasonable extent, while at the same time increasing the flexibility and extensibility of the CSM Testbed.

We have increased the efficiency of GAL by a factor of 2+ and we have made several improvements in CLIP.

For Task 5, Matrix Algebra Methods and Utilities, the goal is to investigate the current capabilities of the CSM Testbed and the required improvements to the CSM Testbed for performing the matrix algebraic operations required for the analysis of present and future CSM focal problems using advanceds methods.

We have made an extensive study of the matrix algebra functions in SPAR that includes documenting these routines and supplying inline comments.

For Task 6, Data Management for Parallel Computers, the goal is to explore and develop concepts for managing structural analysis data on MIMD computers. The ultimate objective is to develop and implement parallel I/O in the GAL database manager used in the CSM Testbed.

We have designed a parallel I/O system based on domain decomposition. Our next step is to implement the design.

In closing we discuss a possible future extention of the CSM Testbed Architecture, a visual user interface. This user interface would increase the effectiveness and efficiency of analysts and developers using the CSM Testbed.

## CSM Testbed Architecture

### Introduction



**Architecture Background**

**Architecture Tasks**

- Near-Term Enhancements
  - Efficiency, Enhancements & Documentation
- Matrix Algebra Methods and Utilities
  - Study, Evaluation & Documentation
- Data Management for Parallel Computers
  - Exploration, Development & Documentation

## CSM Testbed Architecture

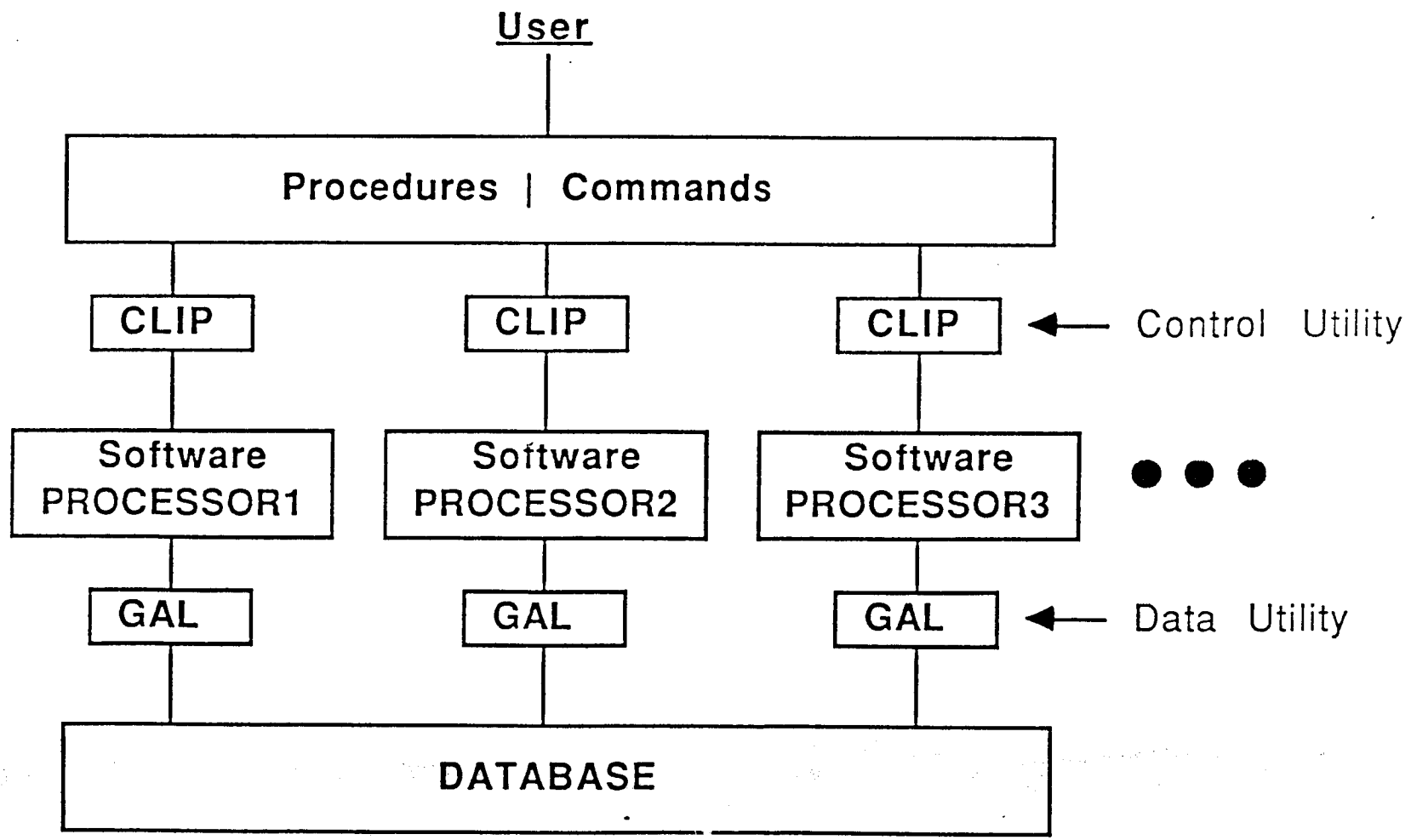
### Introduction

This presentation on the CSM Testbed Architecture includes the background for the CSM Testbed Architecture and a description of the three architecture related tasks under the NASA/Langley CSM contract: 1) Task 2 — Near-Term Enhancements, 2) Task 5 — Matrix Algebra Methods and Utilities, and 3) Task 6 — Data Management for Parallel Computers. For each task the objectives, subtasks chosen to achieve the objectives, and the accomplishments are presented.

# CSM Testbed Architecture

## Architecture Background





## CSM Testbed Architecture

### Architecture Background

The CSM Testbed Architecture is based on the integrated software system called NICE (Network of Interactive Computational Elements). NICE began as a long-term software-research project in 1980 at the Lockheed Palo Alto Research Laboratory [1],[2]. NICE consists of a generic architecture that implements what computer scientists call a "virtual machine," and of independently-executable, application-specific processors. The architecture consists of three software components that serve all processors: 1) the command language interpreter CLIP, 2) the global database manager GAL, and 3) miscellaneous tools to pre-process, scan and maintain source code.

The processors perform the useful work and communicate through a global database that survives processor execution. A key point is that NICE has no central executive. Each processor, being independently executable, has its own main program and its own command-interpretation shell. Operational compatibility is achieved by replicating two architecture components: the command interpreter CLIP and the data manager GAL, in each processor.

In the original CSM Testbed the processors are from SPAR [3]. They have been augmented by Processors and Procedures developed under the CSM contract to give the CSM Testbed non-linear solution capability.

## Near-Term Enhancements

### Objectives & Subtasks



**Objectives**

- Quickly correct inefficiency in the GAL database manager
- Quickly correct deficiency in the CLIP command language interpreter

**Subtasks**

- GAL-DBM Efficiency
  - Improve TOC Search
  - Investigate Local Memory Management
  - Redesign of GAL Database File Structure
- CLIP Enhancements
  - Clean Up Deficiencies
  - Implement Simple Variable Macrosymbols
  - Enhance Macrosymbol Hashing Techniques
- Improve Documentation
  - Deliver and Present Immediate-Term Enhancements
  - Write Draft NICE Application Programmer's Tutorial
  - Write Draft NICE Quick Reference Document
  - Revise NICE Documentation

## Near-Term Enhancements

### Objectives & Subtasks

For Task 2, Near-Term Enhancements to the CSM Testbed Architecture, the primary objectives are to quickly correct: 1) inefficiency in the GAL data manager, and 2) deficiency in the CLIP-interpreted command language, CLAMP. The corresponding modifications should preserve upward-compatibility to a reasonable extent, while at the same time increasing the flexibility and extensibility of the CSM Testbed.

The subtasks chosen to achieve the objectives are:

Subtask 1 — GAL-DBM Efficiency: To improve the efficiency of GAL a new TOC (Table of Contents) search algorithm will be implemented. Additionally, local memory management and a redesigned file structure will be investigated as possible ways to increase efficiency.

Subtask 2 — CLIP Enhancements: To improve the functionality and efficiency of CLIP we will clean up some functional deficiencies, implement simple variable macrosymbols to speed up loops, and implement improved hashing of macrosymbols to speed look up of macrosymbols.

Subtask 3 — Improve Documentation: To improve documentation we will write a tutorial, quick reference guide, and document any changes in the architecture.

## Near-Term Enhancements

### Accomplishments



GAL-DBM: Improve TOC Search

- New Hashing
  - Non-masked Names
  - Masked Names
- Factor of 2+ Speed Up

CLIP: Clean Up Deficiencies

(Improve Functionality, Documented — Not Implemented, ...)

- Repaired Scoping of Local Macrosymbols
- Implemented SCOPE Phrase Qualifier in *DEFINE Directive
- Implemented Macrosymbol ↔ GAL Directives
  - *MAC2GAL or *M2G — Write Macro to GAL
    - ▷ Replaces several directives with one directive
  - *GAL2MAC or *G2M — Read GAL into Macro
    - ▷ Replaces several directives with one directive

CLIP: Implement Simple Variable Macrosymbols

- Work in Progress
  - Modest efficiency gains — 5% maximum

## Near-Term Enhancements

### Accomplishments

We have implemented a new hashing scheme to improve TOC search in GAL. This produced a factor of 2+ speed up. This is a significant increase in efficiency.

We have modified CLIP: 1) so that the scoping of local macrosymbols is as documented, 2) to activate the SCOPE phrase qualifier in the *DEFINE directive, 3) to implement two new directives, *MAC2GAL and *GAL2MAC, to directly communicate macrosymbol values to the GAL database and GAL database values to macrosymbols, and 4) to affect small changes to the operation of CLIP to support the applications and methods tasks.

We have partially implemented the simple variable macrosymbol. Initial testing indicates this feature will not increase efficiency.

## Near-Term Enhancements

### Timing Results (Sun3/160)



Empty do-loop

```
*do $i = 1,1000
*enddo
```

| Code Version | CPU-secs. |
|---|---|
| CSM-Baseline * | 41.940 |
| CSM-Simple Vars. | 40.300 |
| CSM-Simple Vars.+ | 39.460 |

Do-loop with 1 card image

```
*do $i = 1,1000
 <$i>
*enddo
```

| Code Version | CPU-secs. |
|---|---|
| CSM-Baseline | 50.700 |
| CSM-Simple Vars. | 84.860 |
| NICE-Baseline † | 75.200 |
| NICE-Compiled | 15.520 |

Do-loop with multiple card images and directives. (100 times thru loop)

| Code Version | CPU-secs. |
|---|---|
| CSM-Baseline | 132.980 |
| NICE-Baseline | 131.320 |
| NICE-Compiled | 17.100 |

* CSM-Baseline is NICE with Boulder modifications

† NICE-Baseline is NICE without Boulder modifications

## Near-Term Enhancements

## Timing Results (Sun3/160)

The first group of timing results show that roughly 5% efficiency improvement can be attained by implementing simple variable macrosymbols for an empty do-loop. This is not a significant improvement, especially if you consider more realistic cases.

The second group of timing results are performed for the simpliest realistic case. Here we see the simple variable macrosymbol implementation (CSM-Simple Vars.) falls down because the symbol **$i** is no longer stored in card image form and must be converted at every iteration. Please note that, this time represents a quick first attempt, we should be able to improve on this. However, the simple variable macrosymbol implementation will probably only be able to match the baseline results at best. Comparing CSM-Baseline with NICE-Baseline shows the effect of changes made to the CSM Testbed at Boulder in May, 1987. For this simple do-loop the efficiency is significantly improved.

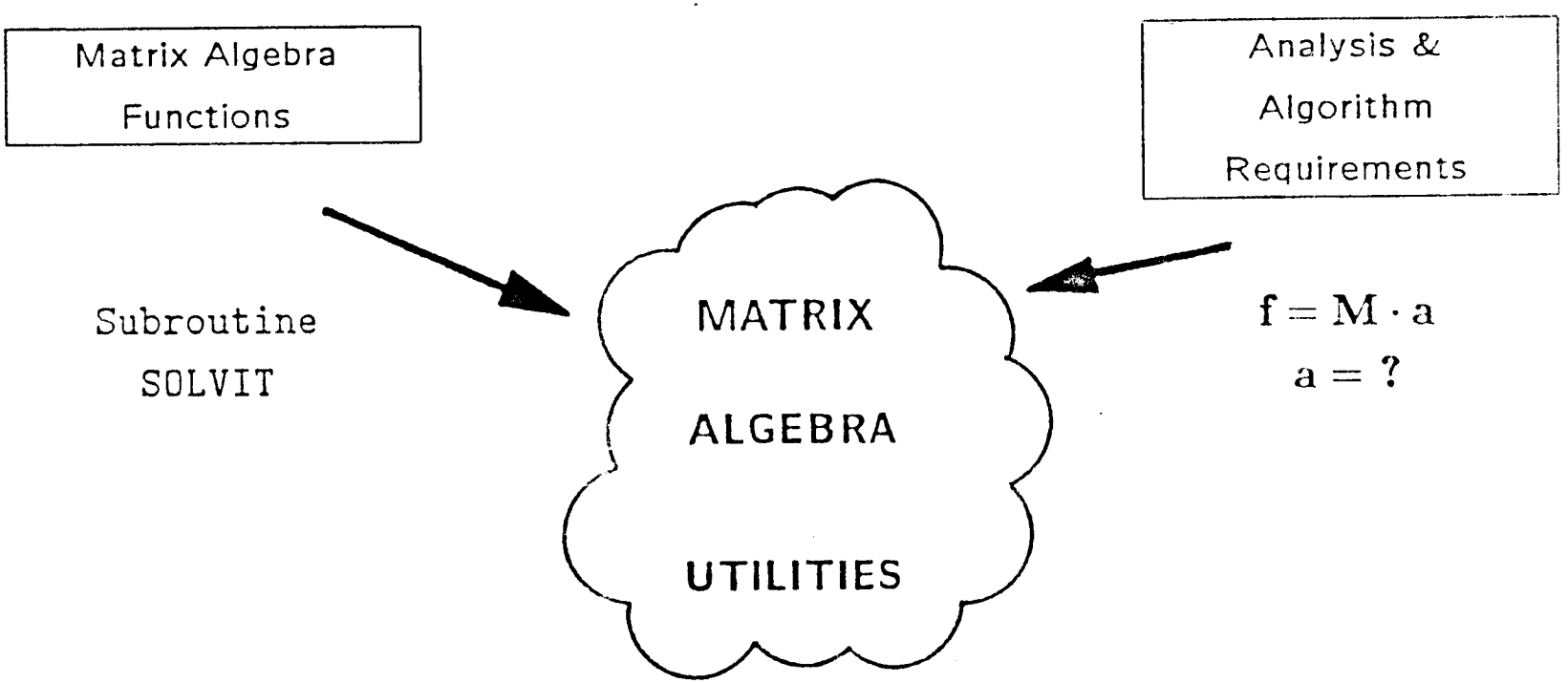
Under the Lockheed Independent Research Program we are pursuing an alternate approach to improving loop efficiency in CLIP. This involves compiling the do-loop by translating it into FORTRAN, then compiling and re-linking. Note that, the NICE-Compiled result shows significant improvement.

The third group of timing results are performed for a typical do-loop found in applications Procedures. Note that, the modifications to enhance simple loops (CSM-Baseline) no longer shows an improvement against NICE-Baseline. Again the compiled version shows a significant efficiency improvement.

The NICE-Compiled design is still undergoing tests and there is much work to be done before it would be ready for implementation in a production environment. However, it appears to be a fruitful direction for research and development.

## Matrix Algebra Methods and Utilities





## Matrix Algebra Methods and Utilities

For Task 5, Matrix Algebra Methods and Utilities, the goal is to investigate the current capabilities of the CSM Testbed and the required improvements to the CSM Testbed for performing the matrix algebraic operations required for the analysis of present and future CSM focal problems using advanceds methods.

This initial effort includes a study of matrix algebra routines available in SPAR[3], an evaluation of matrix data structures for advanced algorithms, and documentation.

## Matrix Algebra Methods and Utilities

### Subtask 1

*Lockheed Palo Alto* *CSM Testbed Architecture*

---

Study SPAR Formatted Matrix Data Structures and Processors

- Details of the data structures.
- Interdependence of data structures.
- Data-blocking logic in the data structures.
- Processor data flow and architecture.
- Processor algorithmic logic flow.
- Documentation

## Matrix Algebra Methods and Utilities

### Subtask 1

Under this subtask the current sparse matrix data structures and the Processors that use them in the CSM Testbed will be studied to determine and document: 1) details of the data structures, 2) interdependence of data structures (e.g., K and KMAP), 3) data-blocking logic in the data structures, 4) Processor data flow and architecture, and 5) Processor algorithmic logic flow.

Inline documentation of SPAR matrix algebraic functions in Processors TOPO, K, INV, SSOL, EIG, and AUS will be provided. Enhancements to the documentation for the current SPAR matrix data structures (e.g., K, KMAP, AMAP, etc.) will also be provided.

## Matrix Algebra Methods and Utilities

### Subtask 1 — Progress

*Lockheed Palo Alto* *CSM Testbed Architecture*

| Processor | Data Structures Decoded | Source Code Deciphered | Source Code Commented | Documentation Written |
|---|---|---|---|---|
| TOPO | ✓ | ✓ | ✓ | |
| K | ✓ | ✓ | ✓ | |
| INV | ✓ | ✓ | ✓ | |
| SSCL | ✓ | ✓ | In Progress | |
| AUS (SUM PROD) | ✓ | In Progress | In Progress | |
| EIG | ✓ | ✓ | In Progress | |

## Matrix Algebra Methods and Utilities

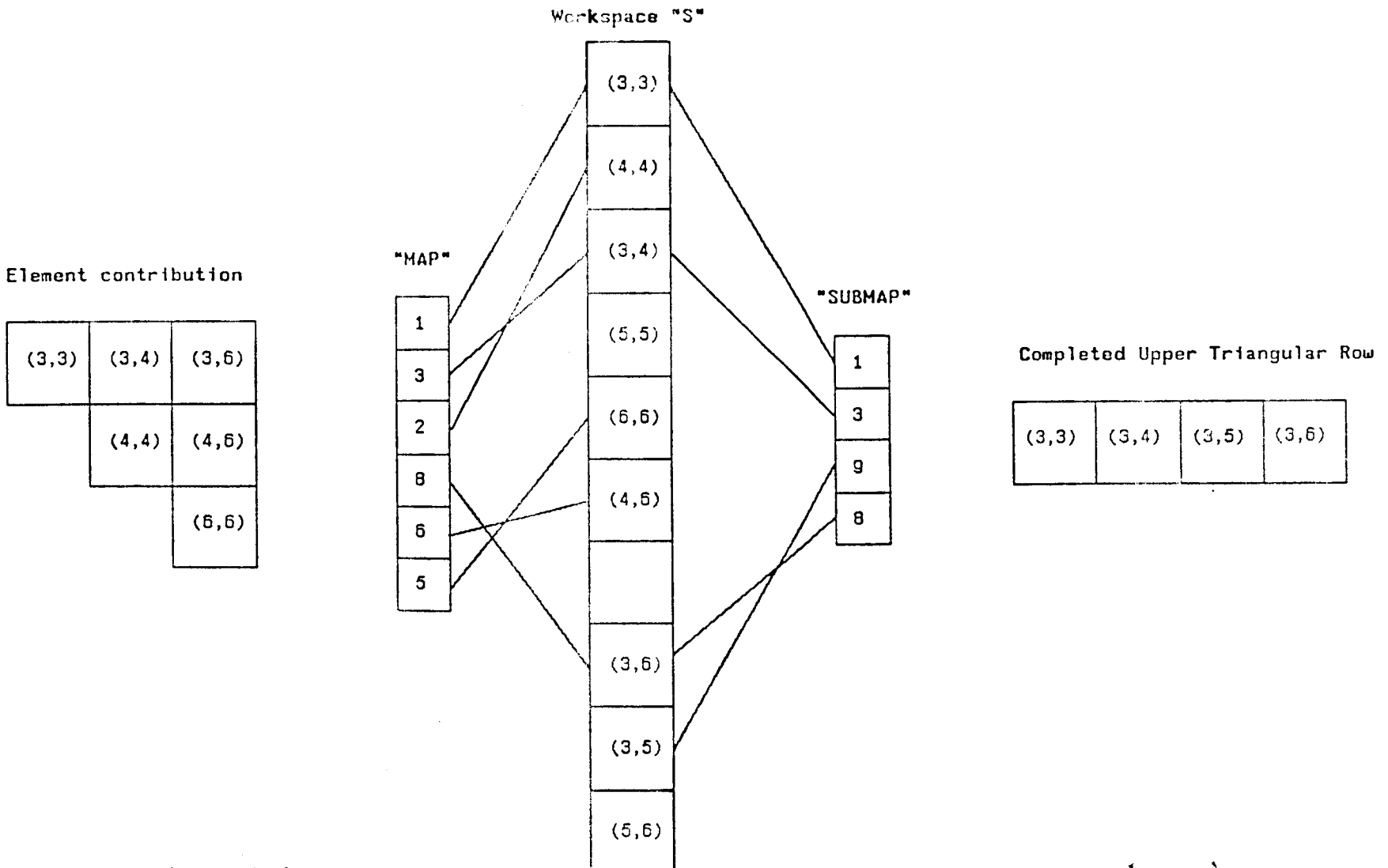
### Subtask 1 — Progress

Work on subtask 1, investigation of current capability in the CSM Testbed for matrix algebra operations, is well underway. We have gained a good understanding of how the sparse matrix processes of topology determination, assembly, factorization, solution, and eigenanalysis are carried out in the Testbed by examining the corresponding Processors: TOPO, K, INV, SSOL, and EIG. The source of each of these five Processors has been deciphered, thus providing detailed knowledge of exactly what each Processor does, the data structures involved, and how the Processors and data structures are interrelated. Matrix summation and multiplication capabilities in the AUS functions SUM and PROD are currently under investigation. Source code documentation is complete for Processors TOPO, K, and INV and in progress for SSOL, AUS, and EIG.

# Matrix Algebra Methods and Utilities

## Subtask 1 — KMAP Use Example

*Lockheed Palo Alto* *CSM Testbed Architecture*



## Subtask 1 — KMAP Use Example

The KMAP dataset may be thought of as a road map to be used during the assembling of the system stiffness matrix. (The same map is also used to assemble the geometric stiffness and the consistent mass matrices.) KMAP contains explicit instructions for element assembly: when to assemble an element, where the element is located, and where to assemble it into the local workspace. It also contains instructions for the archival of completed upper triangular rows from the workspace: when the row is finished, what non-zero submatrices exist for the row, and where each of these submatrices is stored in the workspace.

The example shows what happens during the assembly of the last element attached to joint 3 in a sample problem. The "MAP" for this element contains a pointer into the assembly workspace area, "S," for each elemental submatrix. Once this element's contribution has been assembled, the upper triangular row of the system stiffness corresponding to joint 3 is complete.

"CONECT" is a list of the joints connected to joint 3. This list corresponds to the non-zero submatrices in this joint's row of the system stiffness matrix (except for the diagonal submatrix, which is implicitly included). "SUBMAP" contains a pointer into the assembly workspace for each submatrix to be written out for this joint. Once these submatrices are written out, their space in the workspace is zeroed out and returned to the available pool.

All of the work in figuring out this road map has been done in Processor TOPO. Processors K, KG, and M simply follow the map. Note that, KMAP is only used for assembly, it is not used to interpret an existing SPAR formatted matrix.

## Matrix Algebra Methods and Utilities

### Subtask 2



**Support CSM Algorithm and Methods Development.**

- Dynamics:
  - Multi-body simulation
  - Transient response analysis
  - Joint contact & damping
- Statics:
  - Nonlinear continuation methods
  - Equivalence Transformation
  - Nonlinear substructuring
- Parallel Processing:
  - Mesh partitioning
  - Matrix factor, solve, etc.
  - Vector operations
  - Eigenvalue analysis

## Matrix Algebra Methods and Utilities

### Subtask 2

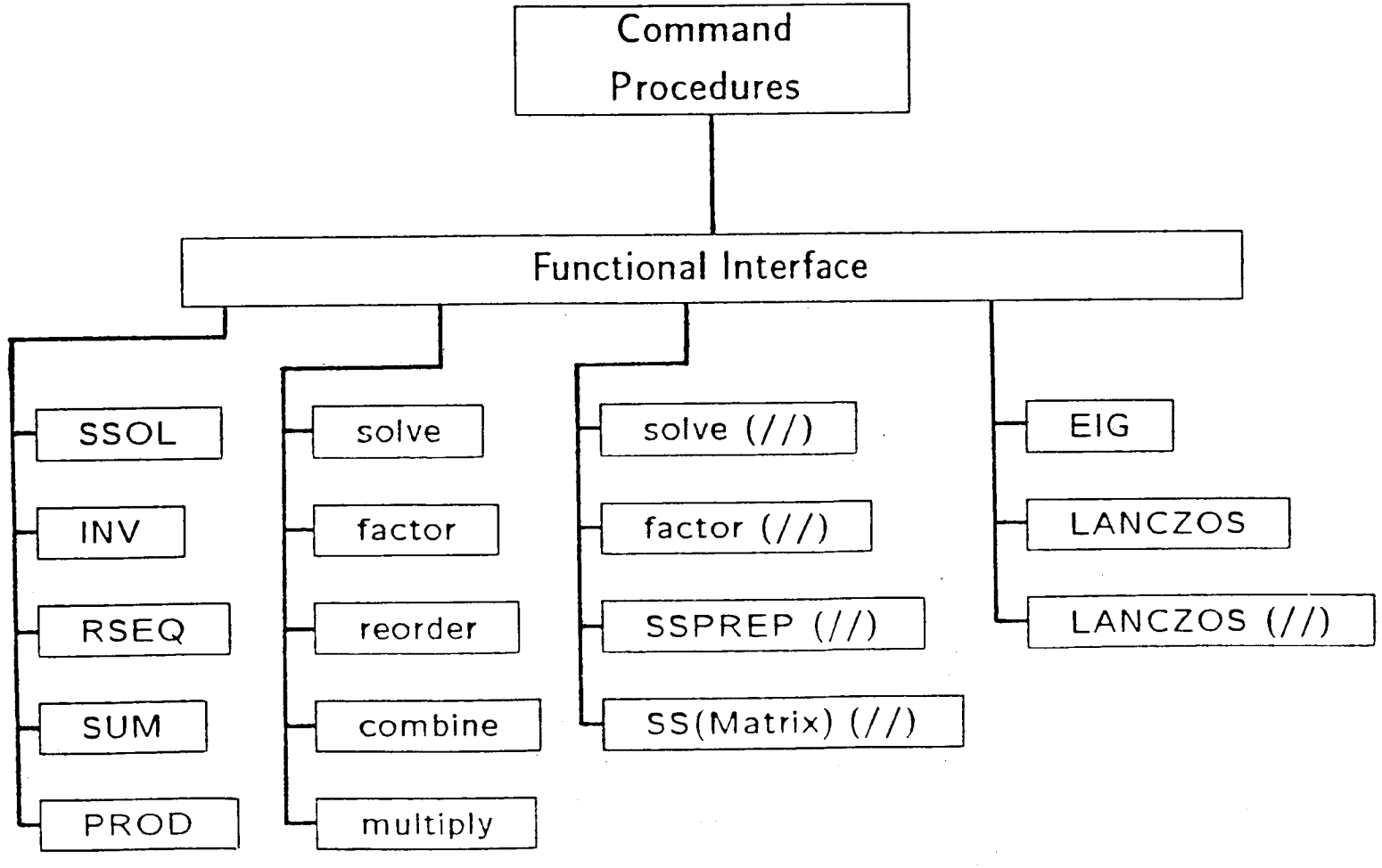
Under this subtask the suitability of sparse and banded (profile) matrix data structures for advanced CSM algorithms will be assessed. The assessment will include the existing CSM Testbed sparse matrix data structure. In addition, a comparison between sparse and profile data structures will be made. Several existing public-domain matrix data structures will also be investigated.

In particular, the following issues will be considered: 1) incorporation of general constraint equations, 2) selective row and column operations for substructuring, 3) selective deletion of active equations for rank corrections as required by analysis algorithms like the Equivalence Transformation and the Riks' method, 4) incorporation of additional equations and node-equation mappings for p- and hp- convergence algorithms, and 5) computational efficiency for a few representative benchmark cases.

## Matrix Algebra Methods and Utilities

### The Ultimate Goal





## Matrix Algebra Methods and Utilities

### The Ultimate Goal

The ultimate goal of this activity is to provide a "complete" set of matrix algebra utilities for developers of new methods and for application to the analysis of advanced CSM focal problems. The matrix algebra utilities should have a breadth from basic operations, such as sum and multiply, to solution packages, such as SSOL and LANCZOS. The implementation of the matrix algebra utilities should be such that the user at the command level can issue commands for a specific operation and have the functional interface determine whether a sequential or parallel execution is appropriate. In addition, the matrix algebra utilities should be available to developers through FORTRAN subroutine calls.

## Data Management for Parallel Computers

### Goal & Subtasks



**Goal**

- Explore and develop concepts for parallel I/O

**Subtasks**

- Develop data management techniques for MIMD computers.
  - Collaborate with CSM grantees in parallel processing.
  - Must be appropriate for finite element structural analysis.
  - Emphasis on shared memory.
  - Modify CSM Testbed architecture for parallelism.
- Make preliminary calculations using techniques developed.
- Present results.

## Data Management for Parallel Computers

### Goal & Subtasks

For Task 6, Data Management for Parallel Computers, the goal is to explore and develop concepts for managing structural analysis data on MIMD computers. The ultimate objective is to develop and implement parallel I/O in the GAL database manager used in the CSM Testbed.

To achieve this goal there are three subtasks.

1) In collaboration with Carlos Felippa, Charbel Farhat, Harry Jordan, and other CSM grantees in parallel processing, develop data management techniques for MIMD computers. The techniques will be appropriate for a finite element structural analysis environment. Emphasis will be placed on modifying the CSM Testbed Architecture to account for parallelism.

2) Make preliminary and exploratory calculations using the techniques developed. Make comparisons between data management requirements for shared memory computers and data management requirements for local memory computers.

3) Present results.

## Data Management for Parallel Computers

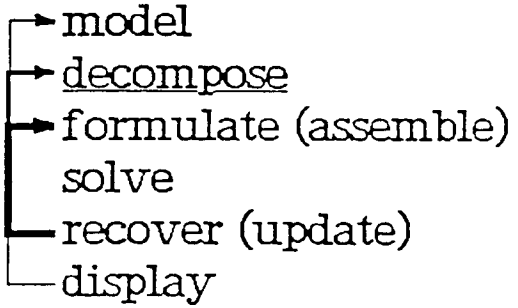
### Develop - Appropriate for FE Analysis



Finite Element Analysis

model problem - graphics ||
decompose domain ||
formulate elements ||
assemble ||
solve (direct & iterative) ||
recover stresses ||
update state ||
display results - graphics ||

Generic




FUNDAMENTAL DIFFERENCE BETWEEN SEQUENTIAL AND PARALLEL COMPUTING IS

**DOMAIN DECOMPOSITION**

## Data Management for Parallel Computers

### Develop - Appropriate for Finite Element Analysis

To develop data management techniques on parallel computers that are appropriate for finite element structural analysis, we first considered how one does finite element analysis on parallel computers. The basic steps in a typical linear, non-linear, static, and dynamic analysis by the finite element method on a parallel computer are: 1) model problem, 2) decompose domain, 3) formulate elements, 4) assemble, 5) solve, 6) recover stresses, 7) update state, and 8) display results [4]. Each step can be computed with parallel algorithms. Thus the finite element method for structural analysis is highly parallelizable.

In a generic sense the finite element structural analysis method on a parallel computer has six steps: 1) model, 2) decompose, 3) formulate, 4) solve, 5) recover, and 6) display. The heaviest arrow indicates the path most used in solving a non-linear or dynamic problem. The heavier arrow indicates that occasionaly the problem changes to the extent that another decomposition is needed. The thinest arrow indicates that at times one looks at the results and starts all over again.

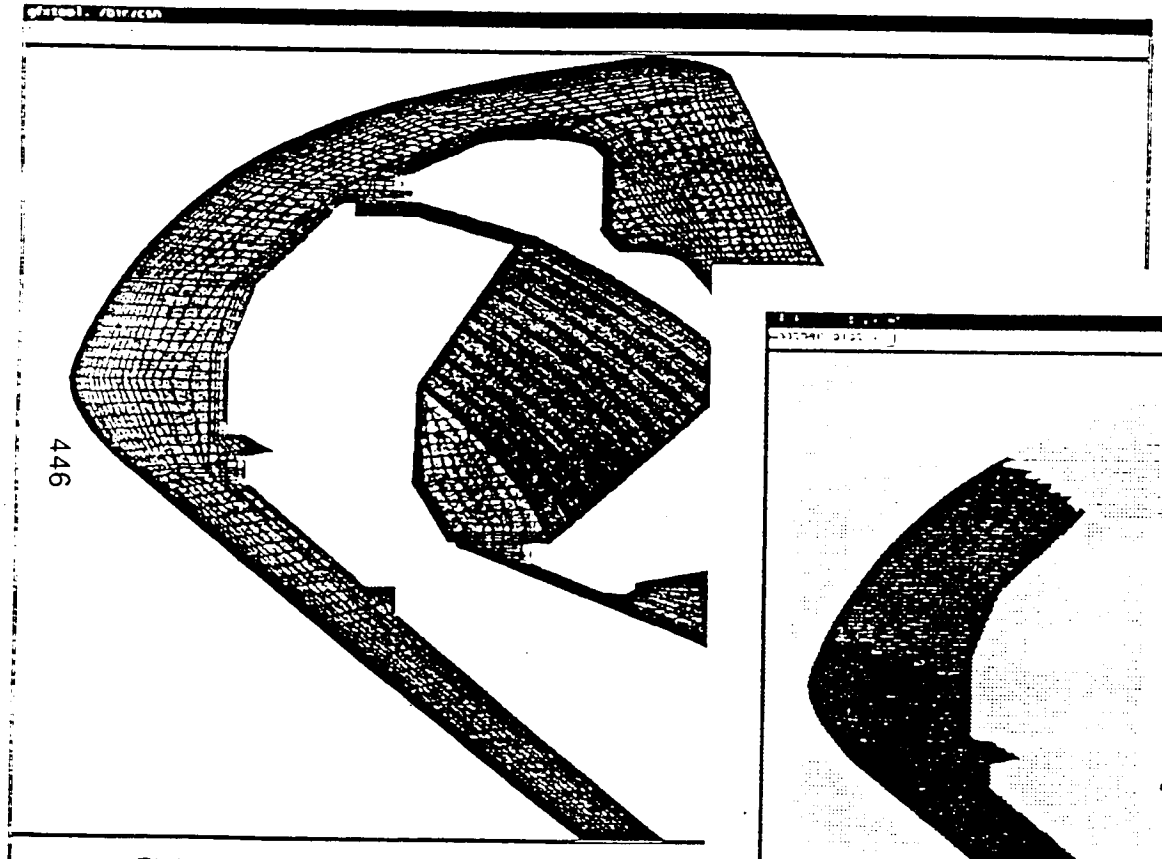
Throughout all of the steps the only step that is unique to the finite element analysis method on parallel computers is that of DOMAIN DECOMPOSITION. Although some algorithms for sequential computers use domain decomposition, algorithms for parallel computers require domain decomposition.

# Data Management for Parallel Computers

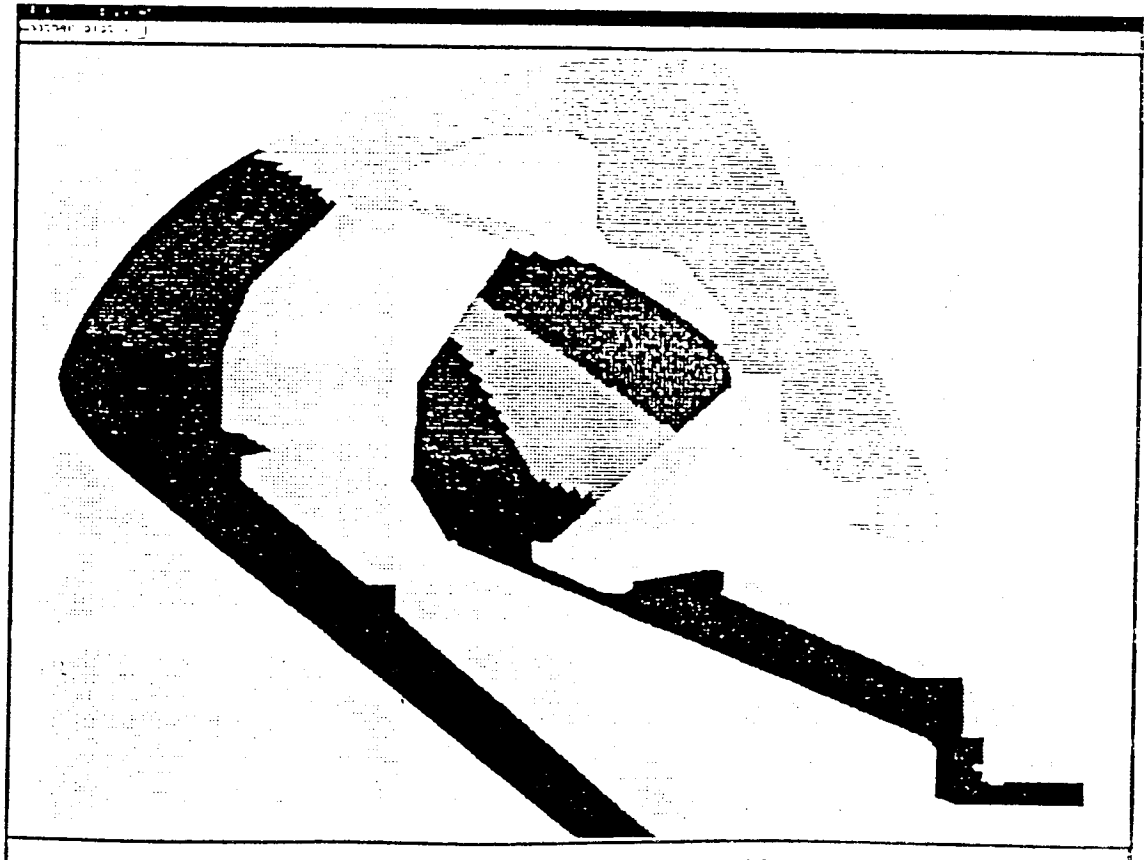
## Domain Decomposition

*Lockheed Palo Alto* — *CSM Testbed Architecture*





Finite element mesh for nozzle problem (6216 e



Directional Bisection mesh partition for nozzle problem ($N_{part} = 8$)

## Data Management for Parallel Computers

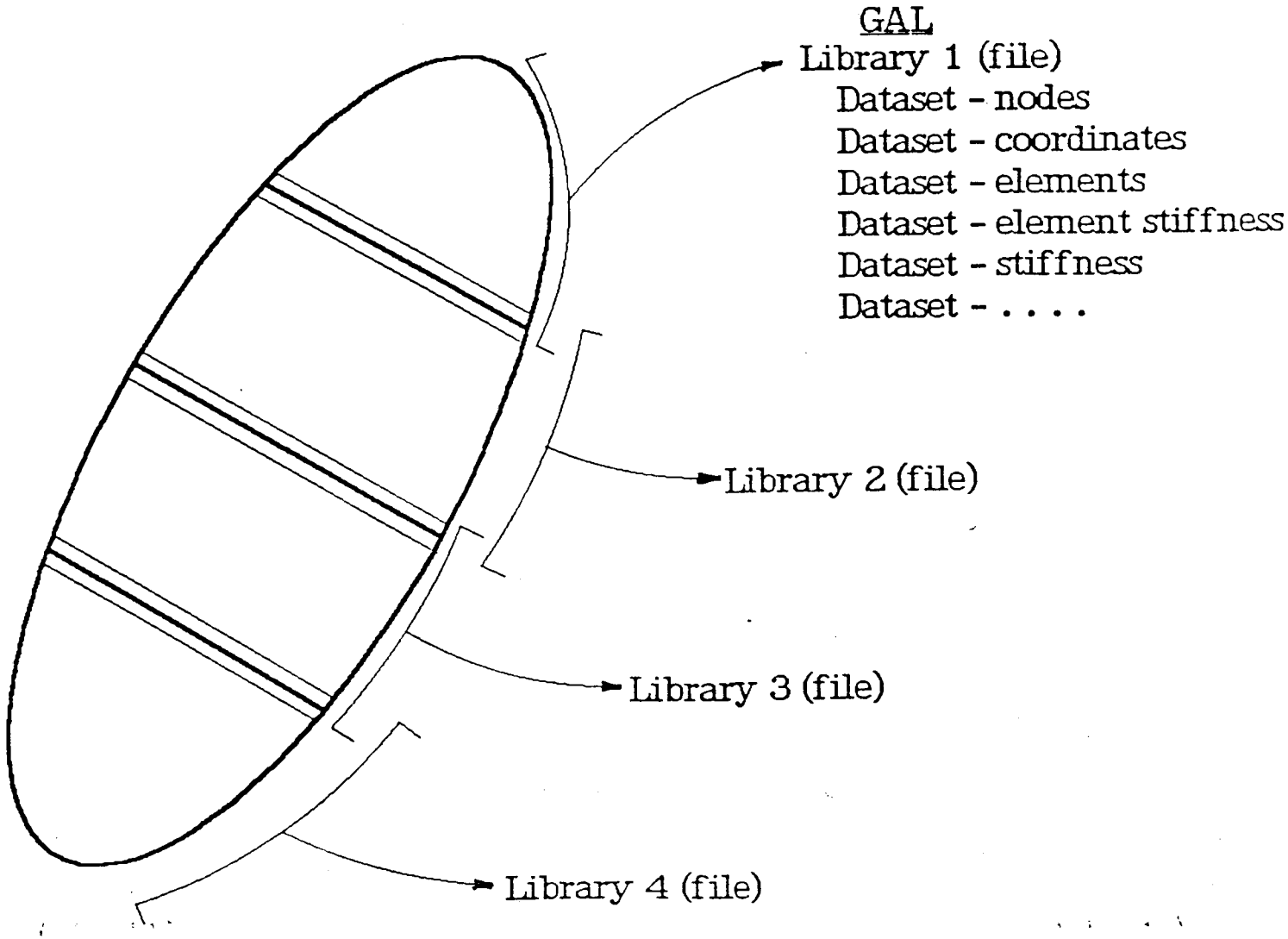
### Domain Decomposition

Our design for parallel I/O is based on the idea of domain decomposition (a data partition based on the physical problem). We have extended the domain decomposition idea for parallel computing to include the amount of memory available to solve a given problem. That is, current domain decomposition algorithms only address the number of hardware processors available; the size of the problem is limited to the physical memory available to each hardware processor. However, note that the domain decomposition process is inherently recursive, in that if you 4 processors a parallel algorithm works just as it would for 8, 16, or ... processors. Thus, we can include the memory requirements for the problem in addition to the number of processors by the following argument. Let's say we have 16 processors but the problem will not fit in the available memory. So we assume we have 32 processors ($2 \times 16$) processors with the same amount of memory available to each as before (i.e., twice the total memory) and say this problem now fits. The problem can be run by making two passes through the 16 processors.

## Data Management for Parallel Computers

### Domain Decomposition — Data Decomposition





## Domain Decomposition - Data Decomposition

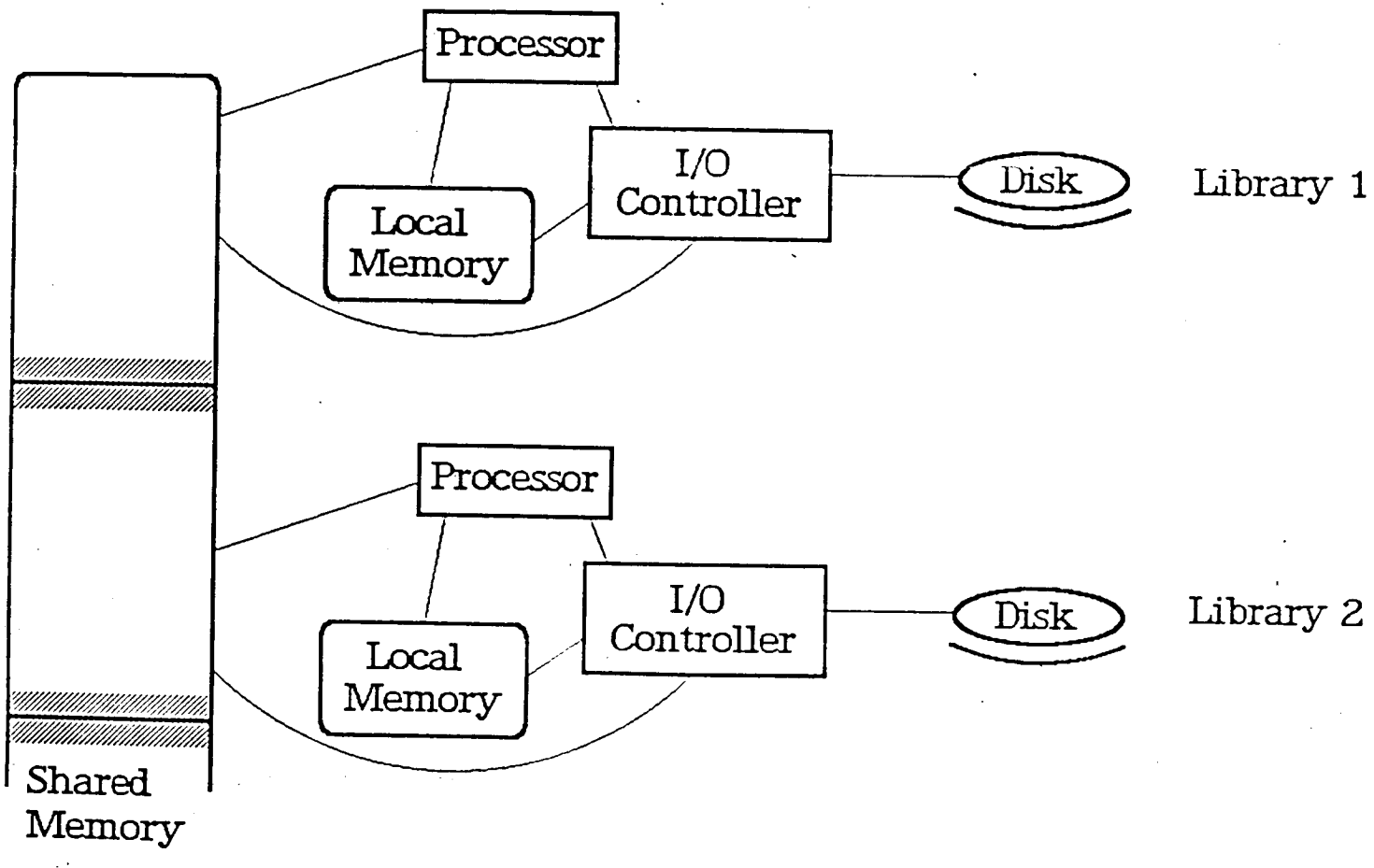
Now how do we use this extended domain decomposition idea to achieve parallel I/O in the CSM Testbed Architecture? As the domain decomposition process is run we create a GAL Library (file) that contains the data for each subdomain. So in the example above we have 32 GAL Libraries. Note that, each set of data also includes replicated subdomain boundary data, as we intend to use a local memory paradigm throughout. This eliminates memory conflict problems that arise in shared memory architectures [5], and allows us to run on both local and shared memory architectures by mimicking a message passing environment on both architectures [6]. The replicated data must be tagged for easy identification, but this and a small additional storage required on the shared architecture are the only overhead.

There are analogues between this design and the window design [7]. In our design each subdomain can be likened to a window on the discretized problem instead of just the data. Thus we achieve the ability to use complex and dynamic data structures through the use of the structures in the GAL database language. Whereas the window design does not appear to easily accommodate data structures other than an array.

## Data Management for Parallel Computers

### Parallel I/O — Hardware

*Lockheed Palo Alto* — *CSM Testbed Architecture*




replicated data (boundary data)

## Data Management for Parallel Computers

### Parallel I/O — Hardware

Here we present an idealized picture of a parallel computer architecture that contains shared and local memory, an I/O controller and disk drive for each processor. On each disk we have a GAL Library (actually for our example above each disk would have two GAL Libraries but only one is active). Most likely, in reality, we would have to share I/O controllers and disk drives among all processors or groups of processors. An implementation of our design will have to consider what is actually available to us. However, the Cray2 and the NCUBE have elements of what is shown here. We believe we can map our parallel I/O design onto real architectures.

Assuming we can do this lets walk through the steps in one iteration (cycle) of computation for our example. We initiate the first 16 subdomain problems on the 16 available processors. Each processor reads the data for its subdomain from its GAL Library. This is done in true parallelism for a computer such as illustrated here. Otherwise the hardware operating system with the CSM Testbed Architecture covering the actual hardware system (virtual machine) will handle the I/O requests to make it appear to be parallel I/O. After the data is read each processor does its computations and writes the data back to its GAL Library. Then the second group of 16 subdomains is processed the same way. At this point the replicated boundary data is updated as is normally done in a message-passing local memory architecture (this is equivalent to synchronization on a shared memory architecture). The GAL Libraries are updated during this process but only the boundary data is effected. Now another step can be taken.

- Employ a local memory execution model.
  - Communicate between processors by passing messages.
  - Decompose computational tasks.
  - Decompose communications (I/O) tasks.

    (Do not mix computation and I/O.)
- Will achieve software that is relatively insensitive to changes in:
  - number of processors,
  - interconnection structure,
  - local or shared memory,
  - amount of memory.

## Data Management for Parallel Computers

### Programming Methodology

The programming methodology we advocate is the local memory execution model. This means message-passing for communications and decomposition of both computational tasks and I/O tasks. We believe this methodology will achieve software that is relatively insensitive to changes in: the number of processors, the interconnection structure of the computer, local or shared memory, and the amount of memory.

Others have advocated this long before us [5] and [6]. Their influence and results seem to indicated they should be listened to. Also, the message-passing environment is used in PISCES [8] and it is part of the original design for NICE [1]. We plan to stay with success.

## Data Management for Parallel Computers

### Next Step(s)



- Implement domain decomposition algorithm in CLAMP.
  - generates separate GAL Libraries from model data
- Modify GAL to accommodate message passing.
  - Send and Receive
  - Queue I/O
- Try out on Cray2 and/or NCUBE.

## Next Step(s)

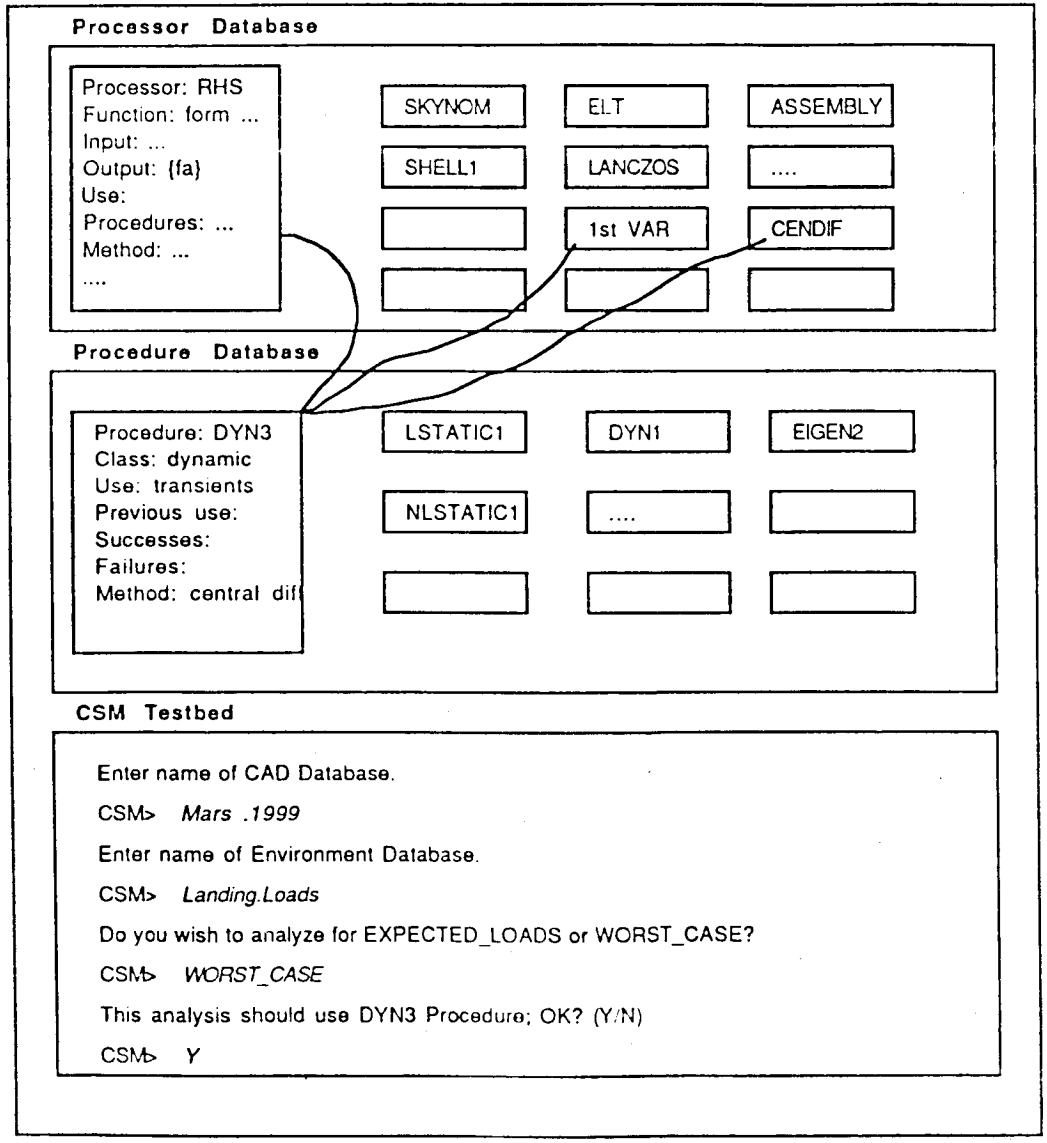
The next steps we plan to take in developing parallel I/O is to implement the directional bisection domain decomposition algorithm [9] in CLAMP. This will generate the separate GAL Libraries for a problem such as that shown in the domain decomposition discussion above. Then we will modify GAL to use send and receive commands. Here we will queue I/O requests as they are received and send an acknowledgement as the requests are processed.

Assuming all goes well we try some real problems on the Cray2 and/or our NCUBE. The FORCE [10] may be used to move some of the CSM Testbed to a multiprocessing environment on the Cray2.

# CSM Testbed Architecture

## Future — User Interface





## CSM Testbed Architecture

### Future — User Interface

The CSM Testbed Architecture supports a rich and complex set of functions and applications. It is often difficult for a beginner to get started or even for an experienced user to be aware of all that is happening or can be done. A visual user interface would provide the support a user or developer needs to effectively and efficiently use the CSM Testbed.

Powerful engineering workstations are becoming common in the research and development environment and will soon be common in the production analysis environment. With standards such as UNIX, X-Windows, FORTRAN77, and ANSI "C" a portable user interface for the CSM Testbed on engineering workstations is doable and desireable.

## References

[1] C. A. Felippa, *Architecture of a distributed processing network for computational mechanics*, **Computers and Structures, 13**, 405-413, 1981

[2] C. A. Felippa and G. M. Stanley, *NICE: A Utility Architecture for Computational Mechanics*, in proceedings of **Finite Element Methods for Nonlinear Problems**, eds., Bergan, Bathe and Wunderlich, Europe-US Symposium, Trondheim, Norway, 1985

[3] W. D. Whetstone, *SPAR Structural Analysis System Reference Manual*, Vol. 1, NASA CR 158970-1, Dec. 1978

[4] C. H. Farhat, *Multiprocessors in Computational Mechanics*, Ph.D. Thesis, UCB/SESM-86/12, Dept. of Civil Engineering, U. of California, Berkeley, CA, Dec. 1986

[5] C. A. R. Hoare, **Communicating Sequential Processes**, §6.3, Prentice/Hall International, Englewood Cliffs, NJ, 1985

[6] S. B. Baden, *Run-Time Partioning of Scientific Continuum Calculations Running on Multiprocessors*, Ph.D. Thesis, LBL-23625, Lawrence Berkeley Laboratory, Berkeley, CA, June 1987

[7] P. Mehrotra and T. W. Pratt, *Language Concepts for Distributed Processing of Large Arrays*, **ACM SIGACT-SIGPLAN Symp. Principles Distributed Computing**, Ottawa, 19-28, Aug. 1982

[8] T. W. Pratt, *The PISCES 2 Parallel Programming Environment*, ICASE Report No. 87-38, NASA Langley Research Center, Hampton, VA, July 1987

[9] B. Nour-Omid and S. N. Glasser, *Recursive Inertial Bisection for Partitioning Finite Element Meshes*, in preparation

[10] H. F. Jordan, M. S. Benten and N. S. Arenstorf, *Force User's Manual*, Dept. of Electrical and Computer Engineering, U. of Colorado, Boulder, CO, October 1986