NOV.20.1997 4:29PM STANFORD UNIV. SPO SATELLITE OFC

NASA/~~TM~~ CR-97- 113004

FINAL
IN-61-R
12IT
124770

Final Report

Development of the Tensoral Computer Language

Grant #NCC2 899

Submitted to:

HPCC
NASA Ames Research Center

Submitted by:

Stanford University
Department of Mechanical Engineering
Thermosciences Division

Principal Investigator:

Joel Ferziger

Associate Principal Investigator:

Eliot Dresselhaus

For the period:

January 1, 1995 through December 31, 1996

Summary

Advanced computer simulations in science and engineering almost universally require the efficient manipulation of huge numerical databases. Such manipulations apply generic mathematical operations — typically calculus and statistics — to large arrays of non-generically represented numerical data. Conventional codes to realize these generic operations must reflect the structure of the particular data they operate on, the numerical methods they employ, and the computer systems they execute upon. Hence, conventional codes — especially those seeking high performance — are highly non-generic. In particular:

- To cope with large scientific databases, data must be *managed*. For example, a data manager may split a large database into pieces small enough to be processed in a single processor's main memory or may spread the data over many processors. Data management issues alone add much complexity to database manipulation and are almost always dependent on the specific computer system and input/output architecture being used.
- Also, the numerical coding of mathematical operations is strongly dependent on both how data is managed and the specifics of how this data is used to represent a given quantity. For example, computing a derivative involves both having the correct data values in memory at one time and employing a numerical method which combines these values to form a numerical derivative.

The research scientist or engineer wishing to perform large-scale simulations or to extract useful information from existing databases is required to have expertise in the details of the particular database, the numerical methods, and the computer architecture to be used. This poses a significant practical barrier to the use of simulation data.

This goal of this research was to development a high-level computer language — called Tensoral — designed to remove this barrier. The Tensoral language provides a framework in which efficient generic data manipulations can be easily coded and implemented:

- Tensoral is *general*. The fundamental objects in Tensoral represent tensor fields and the operators that act on them. The numerical implementation of these tensors and operators is completely and flexibly programmable. New mathematical constructs and operators can be easily added to the Tensoral system.
- Tensoral is *compatible* with existing languages. Tensoral tensor operations co-exist in a natural way with a host language, which may be any sufficiently powerful computer language such as Fortran, C, or Vectoral.
- Tensoral is *very-high-level*. Tensor operations in Tensoral typically act on entire databases (i.e. arrays) at one time and may, therefore, correspond to many lines of code in a conventional language.
- Tensoral is *efficient*. Tensoral is a compiled language. Database manipulations are simplified, optimized and scheduled by the compiler eventually resulting in efficient machine code to implement them.

1 Introduction.

Many computer simulations in science and industry — for example, in computational fluid dynamics, in geophysical modeling, in protein and drug design, in computational electromagnetics — involve the manipulation of huge quantities of numerical data. Databases associated with these simulations are often so large (consider the roughly 10^9 bytes needed for a single variable at a single time on a $512 \times 512 \times 512$ grid) as to make even relatively simple analysis and graphical visualizations of them unwieldy — even on the largest of current supercomputers. The numerical methods of such analysis and simulation codes are significantly complicated by the data management needs of large data-sets and by the need for high-performance on increasingly complex computer architectures.

For such purely technical reasons much useful numerical data becomes underutilized by the scientific and industrial communities. Databases, once produced, are most likely analyzed by only a single research team (those who authored the simulation), at a single site (where the simulation was developed), and from at most a few perspectives (usually only from those most familiar to the simulation's authors). Since future scientific simulations and experiments will produce even larger datasets, involve more complex problems, and be performed on more complex supercomputers (for example, ones having many independent processors) data management issues will become increasingly difficult.

The difficulties of working with and sharing such large scientific datasets are well known to the NASA Ames and Stanford Center for Turbulence Research (CTR) communities. Over the years CTR and Ames scientists have developed many techniques for generating and analyzing large fluid dynamics databases. Part of the CTR charter includes making this data available to the fluid dynamics community. Therefore CTR has often been faced with the difficulties of sharing large databases and the simulation necessary to analyze them.

Motivated by the need to more effectively share data and simulation technology, a program was launched to develop a database and post-processing facility. Such a facility required a simple technique by which generic data manipulations could be applied to a broad range of numerical databases. Thus, the Tensoral language — in which these manipulations may be coded and implemented — was born.

After several years of preliminary development a prototype compiler for the Tensoral language was implemented. This prototype system performed efficient and flexible analysis of turbulence simulation data. The goal of this research was to turn the promising ideas of prototype system — whose use was essentially limited to analysis of turbulence databases — into a generally usable production system.

This report is organized as follows. In section 2 we define the problem at hand: specifically, we define the relevant mathematical operations of database processing (section 2.1) and characterize the numerical data on which these database operations are to be performed (section 2.2). Also, we describe the current state of database processing at CTR (section 2.3) by giving a specific example of how current post-processing codes are written. Having defined the problem in section 2, we outline its solution in section 3. We introduce the Tensoral language in which the post-processing calculus is coded and describe how the Tensoral compiler transforms high-level mathematical database operations into efficient machine code code to perform them. We conclude (section 4) by describing the accomplishments and the current status of the project.

2 The problem: generic calculus on non-generic databases.

Most important quantities in scientific simulations and databases are either tensor fields or are closely related to them. For example, in fluid dynamics simulations one evolves the Navier–Stokes equation to produce a velocity vector field $\vec{u}(\vec{x}, t)$. In fusion or electromagnetics simulations one evolves Maxwell's equations for the magnetic and electric vector fields $\vec{E}(\vec{x}, t)$ and $\vec{B}(\vec{x}, t)$. Other simulation evolve other differential equations to produce other numerical tensor fields.

2.1 Generic calculus.

Much of what has been learned about the behavior of these general equations (i.e. Navier–Stokes and Maxwell) is phrased not in terms of the basic quantities they evolve (i.e. the velocity field $\vec{u}$ or magnetic field $\vec{B}$) but in terms of quantities *derived* from them. Important quantities derived from the velocity $\vec{u}$ in fluid dynamics include,

- the vorticity vector field, $\vec{\omega}(\vec{x}, t) = \nabla \times \vec{u}(\vec{x}, t)$,
- the pressure scalar $p(\vec{x}, t)$ and its gradient ∇p,
- the strain rate tensor, $S = (\nabla \vec{u} + \nabla \vec{u}^t)/2$,
- the mean velocity $\langle u_i \rangle$ profiles,
- the Reynolds stress tensor $R_{ij} = \langle u_i(\vec{x}) u_j(\vec{x}) \rangle$,
- the pressure strain correlation $\langle p S_{ij} \rangle$,

The common thread in all of the simulations mentioned here is the desire to perform calculus and statistics — specifically, derivatives (curl, divergence, et al.), integrals, inverting the Laplace operator ∇^2, averages, correlations, probability distributions, coarse graining, etc — all on numerical data. Also, since the numerical data in question were generated using a particular (i.e. non-generic) numerical scheme for differentiation, integration, averaging, etc., consistency requires that these operations always be performed using this same scheme. Thus, the numerical implementation of calculus and statistics is certain to be non-generic.

2.2 Non-generic simulations and databases.

Although the mathematical operations we seek to perform are generic, the data we seek to perform them on are not. For example, many families of fluids simulations and databases have been developed by members of the research community at NASA, CTR and elsewhere. These simulations represent solutions to either the incompressible or compressible Navier–Stokes equations for various flow geometries. Although these datasets have resulted from roughly the same underlying evolution equations, differences in boundary conditions, Reynolds number, and other flow parameters completely change how these simulations are performed. Underlying numerical dissimilarities this sort make it particularly difficult to post-process databases: programs to calculate, for example, mean velocity profiles or vorticity will be quite different for different simulations.

In particular:

- Some simulations use orthogonal functions (Fourier, Chebyshev, Jacobi, etc.) to represent tensor fields. Derivatives become algebraic operations (in transform space), and integrals and averages are performed using Gaussian integration.

- Other simulations represent fields at certain grid points and calculate derivatives using finite difference methods. Such simulations are often set in complex geometries and grids.
- Various simulations of the same physical processes may use different fundamental variables. For example, some fluids simulations evolve the velocity field; others evolve its curl, the vorticity field.
- The simulations developed at NASA Ames and CTR are performed on several different super-computers (IBM SP2, Cray C90, Intel Paragon, Thinking Machines CM-5). These various architectures encourage the user — for performance reasons — to use machine-specific data management techniques (such as using the "Solid State Disk" (SSD) on the Cray, or spreading a dataset over many processors on the multi-processors and using node-local disks for temporary storage). Also, again for performance reasons, one is led to use machine-specific optimized Fourier transforms, Laplace inverters, and linear algebra routines. Furthermore, databases retain some degree of machine-specificity even at a binary level (e.g. machine binary representation, floating point format, file-structuring, etc.).

2.3 Current post-processing.

Currently all post-processing of turbulence data is done "by hand." That is, for each simulation and for each desired quantity an author must either add the required code to an existing post-processor or develop a specific new post-processor, perhaps using an existing one as a model. If the databases in question were small and simple, either of these options would be straightforward. For the reasons discussed above both of these options involve significant effort and complications.

A simple explicit fluid-dynamic example will illustrate these complications. Suppose we desire to calculate the pressure $p(\vec{x})$ given a velocity field snapshot $\vec{u}(\vec{k},t)$, output of a Fourier space isotropic turbulence simulation (the simplest of our databases). What follows is a recipe for inverting the Poisson equation $\nabla^2 p = -\sum_{ij} \partial_j u_i \partial_i u_j$.

- Read in $\vec{u}(\vec{k})$ in k_x-k_y planes and calculate necessary y derivatives in wave space.
- Fourier transform these derivatives from wave y space to physical y space. This is the first of three sub-transform steps that make up a full three-dimensional Fourier transform.
- Read in data in k_x-k_z planes, still in wave space, and calculate necessary x and z derivatives.
- Fourier transform both x and z derivatives into physical space. At this point all of the required velocity derivatives are in physical space.
- Form the source term $\sum_{ij} \partial_j u_i \partial_i u_j$ in physical space.
- Transform source term, now fully calculated, back into full wave space and invert ∇^2.
- Transform result back to full physical space, again using two passes through the database (one in x-y planes, the other in x-z planes).

Much of the complexity of this example stems from the fact that the complete velocity field is too large to fit into even a super-computer's central memory. Thus, the data must be split into "pencils" or "planes" of one or two dimensional data and brought into core memory and processed in small pieces. In other situations even more steps must be taken to perform a similar computation.

Considering the above example, one can see that a post-processor which computes many quantities can become a quite an involved code in its own right. In fact, for many simulations at NASA, post-processing codes are more complex and difficult to maintain than the simulation codes themselves.

3 The solution: the Tensoral language.

The fundamental goal of the Tensoral language is to eliminate the burden of coding such complex data manipulations from the user of the data. Use of Tensoral hides the details of the data, it's underlying numerical methods, and the computer system from the user of the data.

Tensoral is a *very-high-level object-oriented* computer language designed to ease the *efficient* numerical computation and manipulation of *tensor* fields (i.e. those arising in fluid dynamics, electromagnetics, etc.).

- Tensoral is a *very-high-level* language. Support for arithmetic and calculus on tensor fields is an integral part of the language. Tensor operations generally act on entire tensor fields (usually represented by large arrays of numbers), and tensor notation (mimicking standard mathematical tensor notation) is provided. Tensor calculus and statistics are also built into the language: Tensoral supports differentiation, integration, inversion of Laplacians, gradient, averaging, correlations, etc. A single line of Tensoral consisting of, for example, an average, a derivative and an addition corresponds to many lines of code in a traditional computer language.

- Tensoral is an *object-oriented* computer language: objects (*tensors*) and operations on them (*operators*) are abstract; the implementation of tensors and operators is separated from their use. Exactly *how* tensors are represented on a digital computer is flexibly and generally programmable. Thus, a tensor may be represented on a fixed grid or a variable grid; it may be represented in various coordinate systems; it may be stored partially in primary storage (e.g. memory), partially stored on secondary storage (e.g. disk); tensors may be spread across the many processors of a multi-processor. Operations involving tensors are also completely programmable: for example, different tensor representations imply different techniques for computing derivatives and other calculus operations. This object-oriented approach provides the framework in which database and simulation *authors* can provide objects (i.e. Tensoral tensors and operators) to database *users*.

- Use of Tensoral is *compatible* with existing computer languages. Tensoral variables and statements can freely exist inside another computer language such as Fortran or C or Vectoral. Tensoral code co-exists with the *host* language. As much as possible of the syntax and semantics of the host language (i.e. declaration syntax, variable scoping rules, statements, syntax, etc.) are preserved by Tensoral. Thus, Tensoral users are not forced to learn another complete set of computer language rules. Instead they must learn a much smaller set of rules: how to introduce Tensoral variables, how to operate on them and how to reference Tensoral variables from host language code.

- Tensoral is designed to produce highly *efficient* programs. The Tensoral system is built around a compiler; programs are compiled into lower-level host language programs and eventually into machine code. The host code generated by the Tensoral compiler is nearly as efficient as optimized code generated by hand.

3.1 Tensoral by example.

The best way to introduce a new computer language or software system is by example. To wit, suppose a database user desires to study the evolution of the mean pressure strain term $\langle pS_{ij} \rangle$ for a time-series of isotropic turbulence velocity field databases (call them run1, run2, ...). Such a user has already consulted with an isotropic turbulence *author*, who has provided the databases run1, run2, ... as well as the iso (short for "isotropic") tensor environment for use with the Tensoral system. After consulting with the author's documentation, our hypothetical user enters the following Tensoral program into a file ps.tlc on her computer:

```
Line 1   iso main (int argc, char * argv[]) {
Line 2     iso (velocity) u, (u, rank 2) A, (u, rank 2) S, (u, rank 0) p;
Line 3     int f;
Line 4     for (f = 1; f < argc; f++) {
Line 5       u = <argv[f]>;
Line 6       A_ij = u_i,j;
Line 7       S_ij = 1/2 (A_ij + A_ji);
Line 8       p = -unlaplacian (A_ij A_ji);
Line 9       printf ("%g ", ave_xyz (p S_ij));
Line 10    }
Line 11  }
```

In this example, C is the host language. Line 1 declares a C main function main to use iso tensors. iso is used with this C code to introduce isotropic turbulence tensors. Lines 2 and 3 declare variables: f is to be a standard C integer variable, u is to be an iso velocity field, A to be just like u but having rank 2, etc. Line 4 loops over the file arguments given on the Unix command line. Line 5 reads in a velocity field into the variable u. The next three lines compute pressure p and strain S. Line 9 uses the C standard printf function to print out all nine components of the mean pressure strain. The information provided by the database author as part of the the iso tensor environment defines how to read the velocity field (line 5), possibly including mass storage access and transfer, how to perform derivatives (line 6), how to invert the Laplacian operator (line 8), etc.

It should be emphasized here that the above program is a correct and functional input file for the current prototype Tensoral compiler. To execute this program the user would just compile and run her program ps.tlc using the Tensoral compiler.

As is clear from this example, use of the Tensoral language has insulated the user from the myriad internal details of the simulations and databases. (Recall the description in section 2.3 of what is necessary to compute pressure for isotropic turbulence simulations.) Also, one can see that a Tensoral program need not be changed much to analyze databases originating from different simulations — as long as Tensoral authors have provided descriptions of such databases (such as iso above).

3.2 Tensoral for database authors.

It is clear from the previous example that database authors must have a more difficult job than database users. They must teach the Tensoral compiler *how* to generate host code to realize tensors and mathematical operations on them. Such authors must have a detailed knowledge of both the numerical methods they employ and of the model of computation that the Tensoral compiler presents to database authors. In this section we give a brief account of what database authors must know about the Tensoral system.

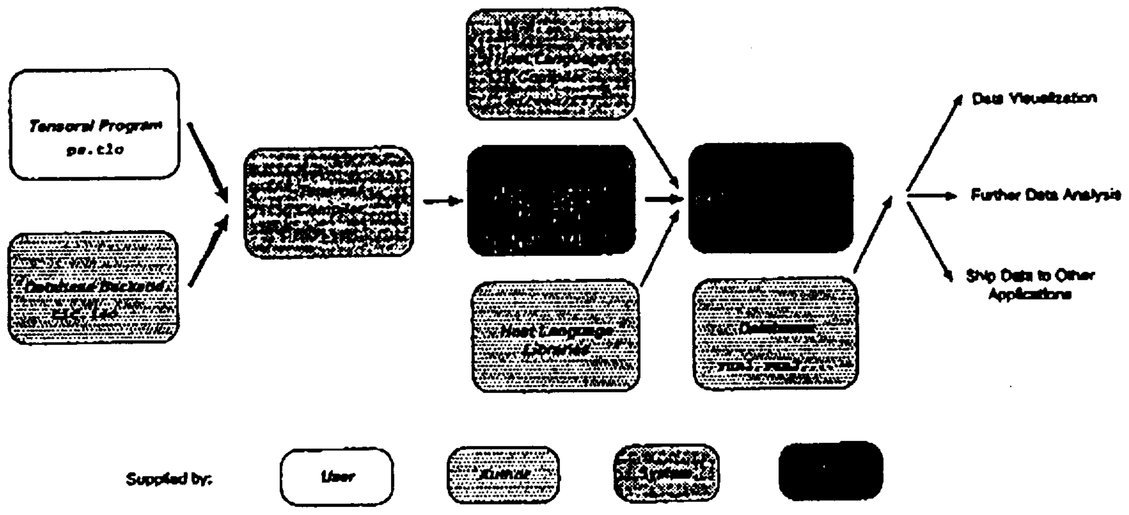

Figure 1: A block diagram of Tensoral compiler use.

Figure 1 illustrates the process in which Tensoral programs are compiled into efficient post-processors to perform the intended task. Following figure 1, the Tensoral compiler

- starts with a Tensoral program supplied by the end database user (e.g. `ps.tlc`),
- determines the required numerical methods and data management techniques in the form of a Tensoral *back-end* (here `tlc.iso`),
- uses the information in this back-end to control the generation of host language code (here `ps.c`) to perform the desired mathematical operations,
- calls the host language compiler to generate machine language output (here, `ps`).
- This compiled file `ps` can then be executed or can be integrated into or called as a subroutine from existing code.

The details of this back-end language are beyond the scope of this document. Suffice it to say that the current prototype isotropic turbulence back-end is sufficient to generate efficient compilations of the above example code and many other test programs.

4 Project accomplishments and future directions.

The project started with improvements and extensions to the prototype compiler. As of by mid-year 1995 we fixed many bugs and had written much of a new back-end for the turbulent channel code. The experience of writing a new back-end was most valuable in highlighting the weaknesses in modularity and coherency of the basic language design.

At this point the project team decided that the Tensoral back-end language needed a complete redesign. In particular, it was decided that the Lisp extension language used to code back-ends was to be abandoned. By the end of 1995 the new system was designed and its implementation commenced.

In the new system, Tensoral — which adds tensor syntax to C— would be implemented in a new coherent and fully modular computer language, called E. E allows for C to be *extended* by adding arbitrary syntax and connecting this syntax with arbitrary code. E distills the basic ideas of Tensoral into a coherent, modular, completely general, computer language. Such a generally useful language was the fundamental goal of this research project.

In 1996 the E compiler framework was designed and implemented. Any computer language translates high-level into low-level. A powerful, and therefore useful, computer language links a powerful and general translation mechanism with powerful and efficient low-level code. Before the E translator was designed, the base language it eventually translated into needed to be designed and coded. In traditional C this base language is essentially provided by the C runtime library; in E this base language is defined by the E runtime library. This runtime library provides the basic low-level code for managing a computer's memory, for handling dynamic arrays, for handling input/output from devices and network, for handling multi-dimensional arrays, for handling syntactic additions, etc. (This low-level library corresponds to the Lisp system embedded in the Tensoral prototype.)

Interim Report

Development of the Tensoral Computer Language

Grant #NCC2 899

Submitted to:

HPCC
NASA Ames Research Center

Submitted by:

Stanford University
Department of Mechanical Engineering
Thermosciences Division

Principal Investigator:

Joel Ferziger

Associate Principal Investigator:

Eliot Dresselhaus

For the period:

January 1, 1995 through December 31, 1995

Tensoral developments in 1995 fiscal year.

The Tensoral computer language and compiler, as it stood at the end of 1994, was novel and functional — capable of generating efficient CFD codes given very-high-level descriptions of the computations to be performed. A user could quickly perform statistical operations, could take derivatives, solve Laplace's equation, etc. simply by entering several lines of code in a computerized mathematical notation. This compact *tensor* notation was translated by the compiler into the many lines of C language code which realized it. Great pains were made to make this automatically generated code efficient.

The 1994 Tensoral, however, was a proof of concept, not a final product. For a new language such as Tensoral to come into wide-spread use, it must more than *improve* on older technology; the new must *revolutionize* the old. Witness that Fortran and C — the dominant languages for scientific and engineering computation for years now — have yet to be dethroned by many fine recent languages. Users will only move to new technology if a commitment to learn it will be well rewarded by time saved in using it. Design vision and dedication to quality were seen as essential for Tensoral to mature from prototype to product.

In the prototype, many decisions were made to save time. The prototype language — although easy to program from a end-user's perspective — was difficult to program on the lower-level. The difficulties of coding Tensoral code generators (*backends*) was enough to dissuade even the most experienced CFD programmers in the NASA Ames/Stanford community. Backends were encoded by Lisp programs which generated C code and interfaced with the compiler's code generator. This approach, although it saved much development time since the Lisp runtime system, data types, and interpreter could be inherited from a standard Lisp system, was sufficient for a research project but not for production language. There were separate languages seen by users (tensors, calculus, statistics, etc.) and programmers (Lisp and C). Tensoral — although perfectly functional — was not a coherent computer language. Furthermore, programming in Tensoral was not modular: it was not clear how common code could be shared between different backends.

During 1995 the faults of the prototype were defined and many were fixed. However, from a backend programmer's perspective the old language was fundamentally flawed. These fundamental flaws motivated a complete re-design of the Tensoral language. The new language distills the successful ideas of Tensoral — high-level computation using specialized notation — into a coherent and modular new language.

The end result of this design was the E language. E, as suggested by its name, aims to be a compact and elegant tool for *extending* C. E extensions to C specify *both* notation and computer code corresponding to this notation. In the new system, Tensoral and its tensor notation and CFD code generation would be implemented as a family of E *extensions* to C. The E compiler itself would also be implemented out of such extensions.

By the end of 1995 the design of the E language was complete and its implementation begun.