S25-61
136899
P.7

N93-17524

Automating FEA Programming

Naveen Sharma
Institute for Computational Mathematics
Department of Mathematics and Computer Science
Kent State University
Kent, OH 44240-0001
Email: sharma@mcs.kent.edu

Abstract

In this paper we briefly describe a combined symbolic and numeric approach for solving mathematical models on parallel computers. An experimental software system, PIER, is being developed in Common Lisp to synthesize computationally intensive and domain formulation dependent phases of FEA solution method. Quantities for domain formulation like shape functions, element stiffness matrices etc. are automatically derived using symbolic mathematical computations. The problem specific information and derived formulae are then used to generate (parallel) numerical code for FEA solution steps. A constructive approach to specify a numerical program design is taken. The code generator compiles application oriented input specifications into (parallel) f77 routines with the help of built-in knowledge of the particular problem, numerical solution methods and the target computer.

Introduction

Engineers and scientists frequently encounter mathematical models based upon partial differential equations (PDEs) in a wide variety of applications. Finite element analysis (FEA) (Zienkiewicz 1980) is a major computational tool for the numerical solution of boundary and initial value problems that arise in stress analysis, heat transfer and continuum mechanics of all kinds. The problem domain is first *discretized* into a suitable *mesh of elements*. Then well-selected analytical approximations are used for solution within each element. The global solution for all discrete points (element *nodes*) of the mesh is computed by numerical iterations taking into account inter-element interactions and boundary conditions.

Simple FEA applications can be performed with canned packages such as NFAP (Chang 1980) and NASTRAN. Situations involving complicated boundary conditions or element properties, non-linear material properties, require customizing many aspects of FEA. In such cases, the finite element solution process consists of a symbolic computation phase followed by a numerical computation phase. Depending on the problem at hand, the symbolic computation phase may involve *construction and analysis of solution approximations*, *simplification of large analytical expressions*, *changing variables and/or coordinates to simplify the problem*, *operating on matrices and tensors with symbolic entries*, as well as *integration and differentiation of analytical expressions*. Results of the symbolic computation phase are then used to construct numerical programs.

Frequently the mathematical models and related computer programs are revised during research, engineering and production. Numerical convergence problems may also require that a different numerical procedure be used for FEA solution steps. When the models are three-dimensional, or use large data sets, the program execution speed is critical. Writing programs for parallel computers to speed-up execution is indeed not a trivial task for modelers. Also, parallel programs written for a parallel machines can not be ported to other machines without significant re-programming effort. State of the art *parallelizing compilers* (Kuck 1978), (Allen & Kennedy 1985) take an existing (sequential) code as input and can produce programs for the target parallel machine. However, these compilers parallelize scientific and engineering applications on the model of linear algebra and either completely ignore the *domain specific* parallelism naturally present in the problem or query the user during compilation.

In recent years, there has been an increase in research and development efforts to alleviate these problems. Existing approaches combine symbolic and numerical computing in various ways. These (coupled) symbolic-numeric systems generally take the user input in a very high-level form and automatically generate numerical code in a procedural programming language like f77 or C for the target computer. Some notable recent projects are Ellpack (Rice, Boisvert, and Ronald 1985), Sinapse (Kant et al. 1990), Alpal (Cook 1990), PDEQSOL (Hirayama, Ikeda, and Sagawa, 1991), (Pe-

[0] Work reported herein has been supported in part by the Army Research Office under Grant DAAL03-91-G-0149

skin 1987), and (Steinberg and Roache 1990). Many of these projects have adopted *finite difference* solution method for PDEs.

Our Approach

We have been working for a number of years (Wang 1986), (Sharma and Wang 1988a), (Sharma 1988b), (Sharma and Wang 1990), (Sharma 1991a) in this research direction and our primary PDE solution method is FEA. We identify *key* solution steps of FEA which are **compute-intensive** and are **reprogrammed** every time new element formulations or boundary conditions are used. Our approach is to employ symbolic computation to generate sequential and parallel numerical codes for the key FEA solution steps. The code is generated in (the parallel version of) f77 on the target parallel computers (currently include Sequent Balance shared memory and distributed-memory Intel iPSC/860). Based on the user input, quantities such as **element shape functions** and **strain-displacement** matrices can be derived using symbolic mathematical computations. The derived formulas are used to generate numerical code for computing element stiffness matrix, solution of system of equations and other solution steps. The generated code can be readily combined with existing FEA codes. The overall scheme is pictorially depicted in Fig. 1. We are de-

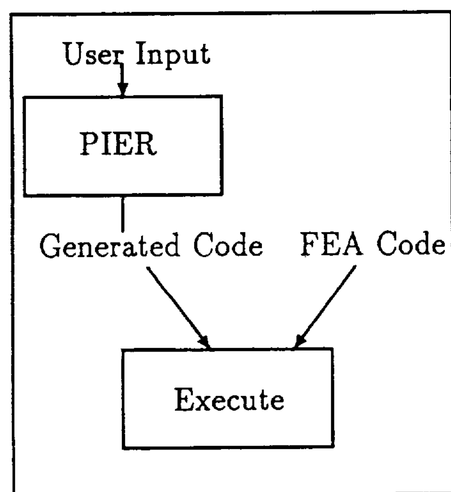

Figure 1: Overview of Approach

veloping a new FEA code generator named **PIER** to build upon our previous work in this area and to break new grounds. PIER is Common Lisp (CL)-based and can work directly with the CL-based MAXIMA. It can be easily ported to other CL-based symbolic computing systems. PIER generates sequential and parallel codes for the key solution steps. In next two sections we briefly discuss our design objectives and PIER's programming knowledge. PIER input specifications and the code generation scheme are overviewed in subsequent sections. We conclude the paper by outlining some relevant issues.

Design Goals

Previous FEA code generators, such as FINGER, P-FINGER and PDEQSOL, are equipped with a fixed number of numerical algorithms for FEA solution steps and the algorithms are parallelized and implemented for one specific parallel computer. Porting the code generator to other machines, thus requires work from scratch. This is a major drawback which should be overcome. Our first design requirement addresses this issue.

> The code generator provides a set of architecture-independent input specifications to design and express numerical algorithms used in FEA solution steps.

In generating parallel programs from the user input specifications, it is possible to take advantage of domain independent parallelism which exists among concurrently schedulable code modules and domain specific parallelism such as carrying out FEA (sub)computations in the *element-by-element* (Winget and Hughes 1985) formulation as opposed to the *assembled* formulation, or substructuring the FE mesh etc. This leads to our second design requirement.

> Automatically generates good implementation mappings (for input specifications) to modern high-performance computers with the help of built-in knowledge of the application domain, FEA solution method, and the target programming environment.

These design requirements allow engineers and scientists to customize the FEA solution process for the desired application area and the problem instances. Only input specifications needs to be altered without worrying about implementation details or the target architecture.

System Overview

PIER provides a knowledge-based programming environment to the modelers. The architecture of the environment comprises following components.

1. **A Programming Knowledge-Base.**
2. **A set of User Input Specifications.**
3. **Code Generator.**

The programming knowledge-base provides generic *Operations* (a set of basic linear algebra computations including matrix-vector product, vector inner product, solving triangular system of equations etc.) and domain specific Operations (a set of basic finite element analysis computations including assembling element stiffness matrices, deriving shape functions, vector preconditioning etc.). A PIER Operation has four

parts: *prologue/epilogue*, a set of *algorithm schemas*, *control dependence graph* (CDG) and the associated *cost-model*. The schemas are stated as templates written in Common Lisp, which include assignments, conventional control constructs, and array/scalar computations. The CDG represents the execution dependence among several sub-computations in the Operation and the cost-model determines the execution cost (computation and communication costs) of the Operation. PIER Operations implement the intended computations in one of the following execution styles:

(S1) `Assembled`: Execute for assembled data.

(S2) `FullyParallel`: Execute for individual element data concurrently.

(S3) `BlockParallel`: Execute for a block[1] of element data.

(S4) `Scalar`: Execute for one element data at a time.

The user input specifications provide methods to specify problem parameters, desired symbolic derivation and combine Operations to construct an FEA algorithm. The code generator generates `f77` programs from the user input specifications for the target architecture. The generated code is compiled and linked on the target machine and executed. The programming knowledge-base also provides completed specifications for frequently used FEA algorithms. The user can, however, specify a new algorithm and add the same to the knowledge-base. The knowledge about programming the target parallel architecture is represented as a set of transformations. These transformations convert CDGs into equivalent `f77` templates. Porting to other computers, thus, require developing the set of relevant transformations.

PIER Input Specifications

One of the major research objectives in PIER is to design a set of very high level input specifications which are used by scientists/engineers as well as system developers to describe FEA computations and problem instances. We advocate a bilingual programming style in which application oriented specifications (i.e. terminology and notations as used in standard FEA texts) can be mixed with regular `f77` syntax to express an FEA algorithm. In designing the PIER input specifications we seek that the specifications should be easy to understand and easy to produce by scientists/engineers and the user specifies only the functionality desired and leaves the implementation details to PIER.

The overall approach is to add statements (which represent domain-specific computations) to `f77`. The set of powerful statements are primarily intended for FEA algorithms. Although this scheme could easily work in other areas of scientific computing. The input specifications support the definition of the element mesh, nodal properties, various data arrays, symbolic derivation and specification of numerical algorithms for the solution procedures. Statements defining storage strategies for FEA data arrays, high-level symbolic/numerical computations (PIER Operations), and straight-line[2] sequences of Operations (PIER *Modules*) can be intermixed with regular `f77` constructs to specify a desired numerical algorithm. We now describe the underlying programming model.

The Programming Model

In general the systematic software development process begins with informal *requirement specifications*. This is followed by one or more than one *design* phases, which define a system structure meeting requirement specifications. The design phase identifies software modules and their organization. The text book style description of numerical algorithms can be expressed at this level of abstraction with relative ease and PIER automates rest of the software development phases i.e. *detail design* and *implementation* for the algorithm. While generating parallel code, the user also specifies the resource constraints (i.e. number of maximum process/processors etc.). The input specifications are hierarchical and the user expresses numerical algorithms in a bottom-up fashion by creating abstractions of higher level in terms of lower ones. This is depicted in the Fig. 2. Let us describe each level briefly.

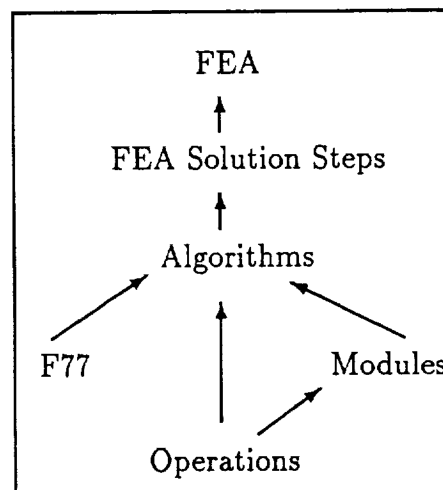

Figure 2: Hierarchy of Computations

- **Operation**: An Operation is the smallest unit of computation (provided in PIER knowledge-base). An operation usually represents a single textbook equation with one variable on the left hand side and an expression involving one or more variables on the right hand side. For example, the equations

$$Temp_1 = r \cdot z$$

[1] A block is a set of elements in which no two elements share a node

[2] No sequence control is involved

$$u = a \cdot p$$

$$Temp_2 = p \cdot u$$

involved in the PCG (Preconditioned Conjugate Gradient) algorithm (Hughes, Ferencz, and Hallquyist 1987) can each be specified by an Operation. An Operation can specify either a symbolic derivation or a numeric computation. For an operation, the variable on the left-hand side is its *output data object*, while those on the right-hand side are its *input data object*. A PIER *Dataobject* specifies numerical values associated with elements/nodes organized in a structured fashion (e.g. matrix, vector).

- **Module**: A Module consists of a sequence of Operations with no entry or exit points except at the beginning and at the end of the module. In other words, control flow enters at the beginning and leaves at the end of a Module. Fig. 3 illustrates Module specifications. In the example, `Module, In, Out, Begin, End` are keywords whereas `VecInnerVec` and `MatTimesVec` represent Operations (for numerical computations).

```
Module CSSA, (In:(a,r,z,p),Out:(Temp1,Temp2))
   Begin
     Temp1 = VecInnerVec(r,z)
     u   = MatTimesVec(a,p)
     Temp2 = VecInnerVec(p,u)
   End
```

Figure 3: PIER Module Specification Example

- **Algorithm**: An Algorithm is specified by combining Modules with f77 constructs. PIER also supplies, from its knowledge-base, certain standard algorithms that can be used directly. Fig. 4 illustrates an example of Algorithm specifications.

```
PIER Algorithm Specification Example
Algorithm:Foo, (In:(a,b),Out:(x))
     Begin
     . . . f77 code . . .
           .
C      Module Call.
    <<(Temp1,Temp2)=Module(CSSA,a,r,z,p)>>
           .
     . . . f77 code . . .
```

Figure 4: PIER Algorithm Specification Example

- **FEA Solution Step**: The breakup of the FEA solution process recognizes eight solution steps. Each step can be solved by more than one (symbolic/numerical) procedure and expects a fixed set of input quantities and computes a fixed set of results. Depending on the problem formulation, the algorithm used for a solution step may be different. Some standard algorithms such as *Gauss quadrature*, *Gaussian elimination* and *preconditioned conjugate gradient* are built into PIER. Others can be supplied by the user through PIER input specifications.

To derive/generate desired FEA computations (symbolic formulae/f77 code) the user must first specify the element mesh, the element properties, the data arrays for material matrix and nodal coordinates. This is followed by the specifications to derive desired element formulae. We now give an example (Fig. 5) where the problem domain is divided into 256 linear triangular elements. Total number of nodes in the mesh is 153. The local degrees of freedom at each node is 1. For complete syntax and detailed examples for PIER input specifications the reader is referred to (Sharma and Wang 1991b).

```
C Defining Triangular Element Mesh.
m = Mesh(Dim:2,Nodes:153,Elements:256)
e = Element(Ldim:1,Nodes:3,Shape:Triangle)
Dataobject x,Name:XNodalCoordinate
Dataobject y,Name:YNodalCoordinate
Dataobject enm,Name:ElementNodalMatrix
Dataobject m,Name:MatMax

C Deriving element approximations.
h=DeriveShape(Algorithm:Polynomial,e)
b=DeriveBMatrix(Algorithm:Displacement,d,h)

C Generating Numerical Code for a FEA Step.
(x)=SolveSystem(Algorithm:Pcg,k,r,File:foo)
```

Figure 5: PIER Input Specification Example

Synthesis Process

In PIER the FEA programs are synthesized by the method of *composition of program components*. The Operations (in PIER knowledge-base), Modules (User-defined) and Algorithms (User-defined) represent program components in the increasing order of hierarchy. The PIER code generator incrementally refines input-specifications into FEA programs. The code generation is overviewed in Fig. 7 and consists of following phases

1. **Parsing Input Specifications**
2. **Problem Definition**
3. **Code Generation**

In the first phase various input specification constructs are identified and translated into PIER internal Common Lisp function calls. The Algorithm template is recognized and preserved. The functions related to the problem definition (parameters of FEA mesh/element and symbolic derivation of element properties) are executed first. This assigns appropriate values to control variables in the environment. The first phase also identifies and analyses PIER Modules, Module Calls and input/output data object specifications. The user-specified element approximations are derived using symbolic mathematical computations (using AKCL-MAXIMA) and the MAXIMA internal representations are translated into equivalent Common Lisp expressions.

The code generation phase generates code for each Module and constituting Operations. Each Module Call in the Algorithm specification generates code for a Module. Symbolic expressions, if any, appearing in the Module body are translated into equivalent Operation specifications. The code for a Module is generated as a set of `f77` subroutine calls and the corresponding subroutines. After generating code for all of the Modules referred to by Module Calls, the Gencray (Weerawarana and Wang 1989) translator is called to translate the generated Common Lisp forms into equivalent `f77` statements and the holes in the Algorithm template are filled appropriately. In the following subsections we describe code generation from Module and Operation followed by an overview of problem solving with PIER.

Module Code Generation

The code generation from PIER Modules is modeled by *flowgraphs*. A flowgraph is a collection of *flownodes*, which represent task instances and directed *edges*, which represent data dependencies among flownodes. The code generator *derives the flowgraph* representation from the sequence of Operations specified in the Module body and *schedules* the flowgraph onto the target architecture. Operations and data objects form flownodes and edges of the flowgraph respectively. The flownodes, thus, represent coarse grained tasks which have unique cost-models and may be assigned different execution styles.

The schedular of the code generator takes as input a flowgraph, a processor count, and Module execution style (optional). The schedular assigns appropriate number of processors and an execution style to each Operation. The execution style must be consistent with the input and output data objects of the Operation. The schedule should conform to the cost-models of Operations and respect the partial order represented by the flowgraph. The overall objective is to produce a schedule with the lowest total cost.

Details of the parallel code generation can be found in (Sharma and Wang 1990) and (Sharma 1992).

Operation Code Generation

Many of PIER Operations involve regular computations and are internally parallelized in the data parallel fashion. Execution styles (*BlockParallel, Assembled* etc.) refers to methods of partitioning the input/output data objects. The user-specified quantities and output of the schedular are used to refine the algorithm schemas, which implements the Operations. PIER accepts input data objects organized in various specialized storage strategies (e.g. Symmetric Matrix, Banded Matrix etc.). The appropriate data reference mapping are automatically generated in the output code.

Problem Solving with PIER

To use PIER in practice, the first step is to prepare a mathematical model describing the physical situation. The modeler, then, prepares the weak statement followed by dividing the problem domain in a series of elements. Here, we are not concerned with the discretization process and assume that one of the several available domain decomposition software tools is used. However, domain discretization data (i.e. element type, nodal coordinates, list of nodes associated with each element) are to be organized in an appropriate fashion for PIER consumption. To derive computations for any FEA solution steps, the modeler must first define the element mesh. This is followed by input specifications for FEA solution steps which include desired quantities/methods for symbolic derivation and numerical computation. If the desired numerical algorithm is not part of the PIER's knowledge-base, the complete algorithm has to be expressed in input specifications. The modelers can use PIER to generate `f77` code for FEA solution steps. The generated code, if desired, can be executed in conjunction with an existing FEA package. The process is outlined in Fig. 6.

Issues

As indicated earlier, in the present work we are focusing on two issues, that we consider critical, in FEA code generation: programmable code generator and code generation for multiple parallel architectures.

Progammable Code Generator

FEA solution method involves symbolic mathematical manipulation and numerical computation with large data sets. To solve a FEA solution step the modelers make choices for the domain mesh, element approximations, and numerical solution algorithm. The choices made are based on: the characteristics of the posed model, target (parallel) computer, and numerical convergence properties. Therefore the FEA solution programs are highly specialized. Code generation systems which would cover all possible cases are bound

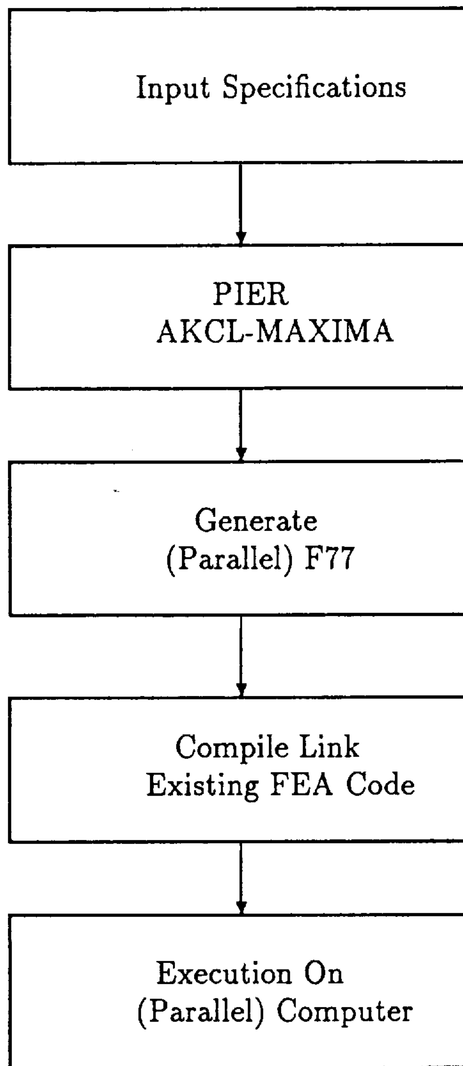

Figure 6: Problem Solving with PIER

to be large, difficult to maintain, and slow. Our approach is to identify and create library of Operations (and possibly Modules), which can generate program components for specific situations. These components are reusable among FEA solution procedures. A solution algorithm can be fabricated using library components and non-FEA specifications in standard f77. The users can customize the generators to their specific needs.

Parallel Code Generation

A major open issue in parallel code generation is the modeling of architecture and the representation of machine specific parallel programming knowledge. In PIER, the computations are represented in an architecture independent formulation (flowgraphs). The Operations generate instances of flowgraphs and attach appropriate code segments to the flownodes. The flowgraph schedular is machine-specific and is the back-end of PIER. Parallel programming rules are transformations from flowgraphs representations to equivalent f77 templates.

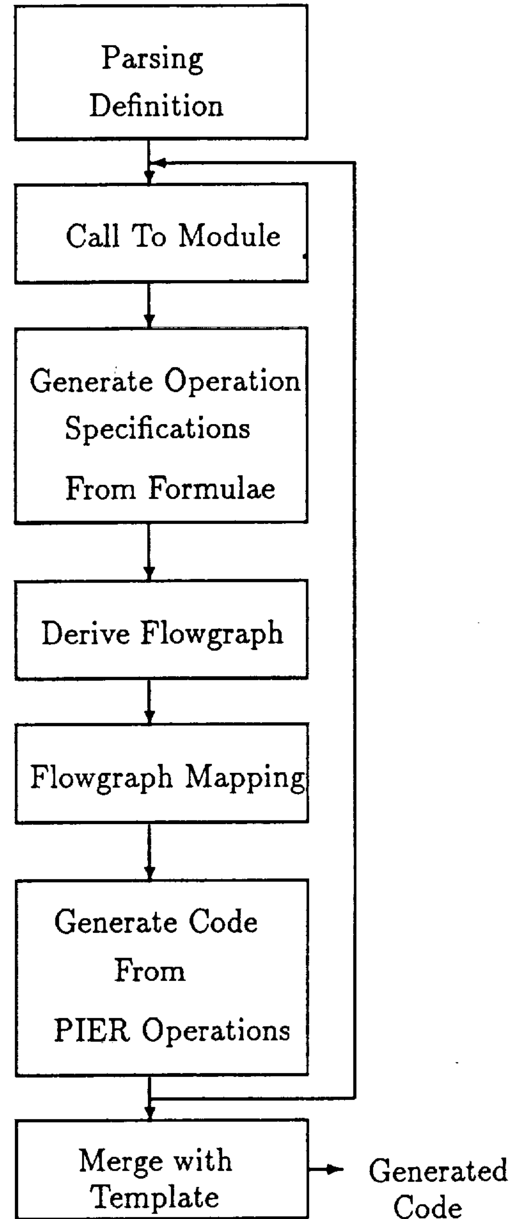

Figure 7: Code Generation Scheme

References

Chang, T. Y. 1986, NFAP - A Nonlinear Finite Element Program, Vol. 2 - Technical Report, College of Engineering, University of Akron, Akron, OH.

Fritzson, Peter and Fritzson Dag, 1991, The Need for High-Level Programming Support in Scientific Computing Applied to Mechanical Analysis, Research Report LiTH-IDA-R-91-04, Department of Computer and Information Science, Linkoping University, S-158 83, Linkoping, Sweden.

Sharma, N. 1988b, Generating Finite Element Programs for Warp Machine, Proceedings of ASME Winter Annual Meeting, Chicago, IL., Nov. 25-28.

Sharma, N., and Wang, Paul S., 1990, Generating Parallel Finite Element Programs for Shared-Memory Multiprocessors, Symbolic Computation and Their Impact on Mechanics, PCP-Vol. 205, *A. K. Noor, I. Elishakoff and G. Hulbert*, Editors, The American Society of Mechanical Engineers, New York.

Sharma, N. and Wang Paul S., 1988a, Symbolic Derivation and Automatic Generation of Parallel Routines for Finite Element Analysis, Lecture Notes in Computer Science, Gianni, P. (Ed.), Proceedings International Symposium on Symbolic and Algebraic Computations 33–56, Rome, Italy.

Sharma N. 1991a, Generating Finite Element Programs for Multiprocessors, Fifth SIAM Conference on Parallel Processing for Scientific Computing, Houston, TX.

Sharma, N. and Wang, P. S. , 1991b, High-level User Input Specifications for Finite Element Code Generation, Conference on Design and Implementation of Symbolic Computation Systems (DISCO), April 13–15, 1992, University of Bath, Bath, UK.

Sharma, N. 1992, The PIER Parallel FEA Program Generator, In Preparation.

Weerawarana, Sanjiva and Wang, Paul S., 1989, Gencray: User's Manual, Department of Mathematics and Computer Science, Kent State University, Kent.

Wang, P. S. 1986, FINGER: A Symbolic System for Automatic Generation of Numerical Programs for Finite Element Analysis, *Journal of Symbolic Computation*, Vol. 2, pp. 305-316.

Steinberg, S. and Roache, P. J., 1990, Using MACSYMA to write finite-volume based PDE Solvers, Symbolic Computation and Their Impact on Mechanics, PCP-Vol. 205, *A. K. Noor, I. Elishakoff and G. Hulbert*, Editors, The American Society of Mechanical Engineers, New York.

Allen, J. R. and Kennedy, K., 1985, PFC: a program to convert Fortran to parallel form, *Supercomputers: Design and Applications*, K. Hwang, editor, IEEE Computer Society Press, pp 186-205.

ParaScope Editor.

Kuck, D. J. 1978, *The Structure of Computers and Computations*, Volume 1, John Wiley and Sons, New York.

Russo, Mark F., Peskin, Richard L. and Kowalaski, A. Daniel, 1987, Using Symbolic Computation for Automatic Development of Numerical Programs. *Coupling Symbolic and Numerical Computing in Expert Systems, II.*

Rice, John R., and Boisvert, Ronald F., 1985, *Solving Elliptical Problems Using ELLPACK*, Springer Series in Computational Mathematics 2, Springer-Verlag, New York.

Kant, E., Daube, F., MacGregor, W., and Wald, J., 1990, Synthesis of Mathematical Modeling Programs. Technical Report, TR–90–6, Schlumberger Laboratory for Computer Science, Austin, TX 78720.

Cook, Grant O. 1990, ALPAL, a Program to Generate Simulation Codes from Natural Descriptions. Technical Report UCRL–102076, Lawrence Livermore National Laboratory, L-35, Livermore, CA 94551.

Hirayama, H., Ikeda, M., and Sagawa, N., 1991, Solution Functions of PDEQSOL (Partial Differential EQuation SOlver Language) for Fluid Problems, In Proceedings of Supercomputing, pages 218-227. ACM Press, November 1991.

Zienkiewicz, O. C. 1980, *The Finite Element Method in Engineering Science*, Mc-Graw Hill, London, pp. 129-153.

Hughes, T. J. R., Ferencz, R. M. and Hallquyist, J.O., 1987, Large-scale Vectorized Implicit Calculations in Solid Mechanics on Cray X-MP/48 Utilizing EBE Preconditioned Conjugate Gradients, *Computer Methods in Applied Mechanics and Engineering*, Vol. 61, No. 2, 1987, pp. 215-248.

Winget, J. M. and Hughes, T. J. R., 1985, Solution Algorithms for Nonlinear Transient Heat Conduction Analysis Employing Element-by-Element Iterative Strategies, *Computer Methods in Applied Mechanics and Engineering*, Vol. 52, pp. 711-815.