High-level User Input Specifications for Finite Element Code Generation

Naveen Sharma and Paul S. Wang*
Department of Mathematics and Computer Science
Kent State University
Kent, OH 44240-0001

January 24, 2003

Abstract

A set of input specifications is designed to enable engineers and scientists, who may not be computer experts, to describe the finite element analysis (FEA) procedures for a new application. The input directs PIER, a code derivation/generation system being developed in Common Lisp, to produce either sequential or parallel FEA codes. The PIER input employs common terminology and notations as used in standard FEA texts and can be intermixed with `f77` statements. A user can transcribe a textbook-like FEA procedure description into PIER input form with relative ease. The input specification supports the definition of the element mesh, nodal properties, data storage scheme, symbolic derivations, and numerical algorithms for the solution procedures. The user can also group high-level statements into *modules* to facilitate parallelization. PIER has an FEA knowledge base which includes two different FEA formulations (element-by-element and assembled stiffness matrix). The design, usage, and implementation of the input specification are described. Examples are given to further illustrate how to prepare PIER input specifications.

1 Introduction and Background

Finite element analysis (FEA) is a major computational tool in engineering and science for the solution of partial differential equations. FEA is frequently used for solving boundary and initial value problems that arise in stress analysis, heat transfer and continuum problems of all kinds. The problem domain is first *discretized* into a *mesh of elements*. Then well-selected analytical approximations are used for

*Work reported herein has been supported in part by the Army Research Office under Grant DAAL03-91-G-0149 solution within each element. The global solution for all discrete points (element *nodes*) of the mesh is done by numerical iteration taking into account inter-element interactions and boundary conditions.

Simple FEA applications can be performed with canned packages such as NFAP [] and NASTRAN []. More complicated situations will involving customizing many aspects of the FEA appraoch. 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.

The iterative process [?] involved in FEA is summarized in Fig. ??. Recent

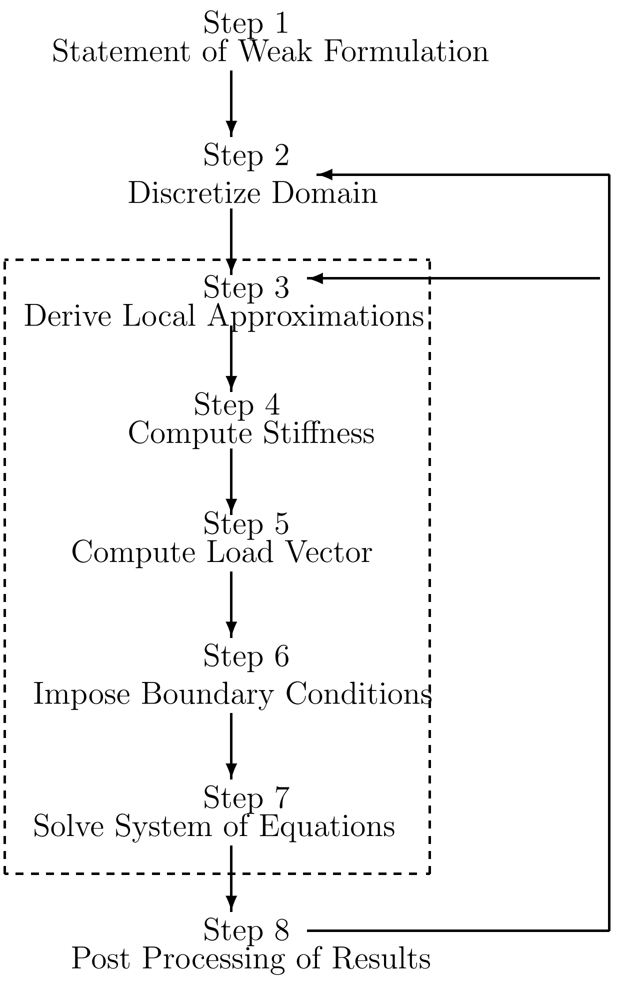
Step 1
Statement of Weak Formulation
Step 2
Discretize Domain
Step 3
Derive Local Approximations
Step 4
Compute Stiffness
Step 5
Compute Load Vector
Step 6
Impose Boundary Conditions
Step 7
Solve System of Equations
Step 8
Post Processing of Results

Figure 1: Problem Solution with FEA

work at Kent [] and elsewhere [] demonstrated the automatic derivation of FEA formulations by symbolic computation. Numeric code can be generated and combined readily with existing FEA packages. This approach can save time and increase efficiency by a wide margin. It has been a prime example of the benefits that can result from a combined symbolic and numeric computation approach.

PIER

We are developing 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. But it is also possible to use PIER with other symbolic computing systems. PIER will generate sequential and parallel codes for the key solution steps (3-7). The code is generated in `f77` possibly with parallel extensions. Based on user input, quantities such as shape functions and strain-displacement matrices can be derived (in steps 3 and 4) using symbolic mathematical computations. The derived formulas are used to generate numerical code for computing element stiffness matrix (step 4), solution of system of equations (step 7) and other solution steps. Of course, generated code can be combined with existing FEA codes. The overall scheme can be pictorially depicted in Fig. **??**. PIER has many advantages and features over existing FEA code generators.

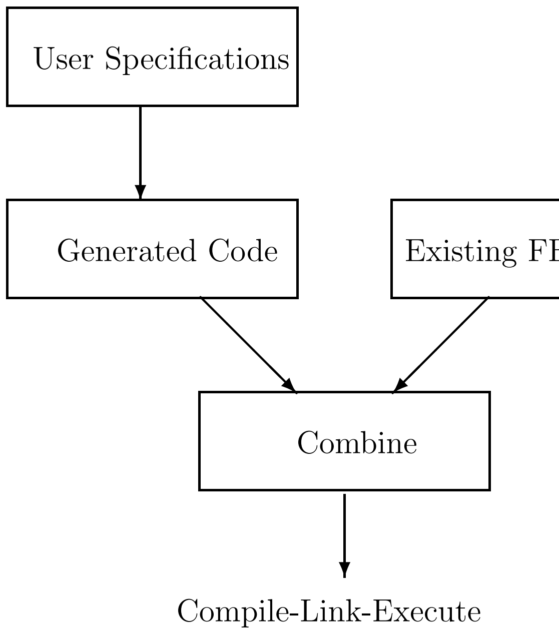

Figure 2: Overview of the Code Generation Approach

Here we focus on its user input specification.

Previous FEA code generators, such as FINGER, lack an easy and flexible way for a user to specify the FEA procedure to be generated. This is a major drawback which must be fixed.

Thus, the PIER input must be easy to specify and flexible enough to express a wide class of FEA solution procedures. The PIER input employs common terminology and notations as used in standard FEA texts and can be intermixed with `f77` statements. A user can transcribe a textbook-like FEA procedure description into PIER input form with relative ease. The input specification supports the definition of the element mesh, nodal properties, data storage scheme, symbolic derivations, and numerical algorithms for the solution procedures.

The user can also specify *modules* of computational statements and attach properties to each statement, forming a structure called a *flownode*. The flownode helps PIER generate parallel code. The code translator Gencray [**?**] is used to convert PIER internal data forms into the *f77*.

We describe in detail the design, usage, and implementation of the PIER input specifications. Although the techniques are described in the context of FEA computations, we feel they are general enough to be useful in a wide range of applications.

2 User 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. In designing the PIER user input specifications we seek the following properties

1. The specifications should be easy to understand and easy to produce by scientists/engineers.

2. The specifications should be precise and should leave no doubt as to the behavior of the system.

3. The user specifies only the functionality desired and leaves the implementation details to PIER.

4. The input structure should be closely related to the user's mental model of FEA and of the PIER system itself.

The input specifications proposed here are problem-oriented and are similar to actual technical language used in FEA textbooks. The overall approach is to add statements to `f77`. The set of powerful statements are specifically tagreted for FEA. Although the scheme could easily work in other domains.

In PIER, the input specifications are intended primarily for expressing numerical algorithms that are used in FEA solution steps. The specification mechanism is not only used by users but also by PIER implementors (or system developers) to build knowledge into PIER. For example the Gaussian quadrature and preconditioned conjugate gradient algorithms are specified in PIER this way. (EXAMPLE of Gaussian quadrature specification???)

PIER specifications are used to describe element mesh, define various data arrays, *instantiate* PIER-defined (YES or NO) FEA computations, parallelization parameters, and to flexibly construct FEA procedures for code generation. Statements defining

- storage strategies for FEA arrays,
- symbolic operations,
- high-level numerical computations/algorithms (YES OR NO),
- *modules* consisting of operation sequences,

can be intermixed with regular `f77` constructs to specify the desired FEA procedure to be generated.

We now describe the underlying programming model.

2.1 The Programming Model

The textbook style description of FEA algorithms state the computational modules and their organization in the generated code. The user give high-level descriptions leaving many details for PIER to take care. In case of generating parallel code, the user also specifies the number of maximum process/processors for the target parallel machine. (Sharma: this is important, you don't want to regenerate the code every time you try the program with one more process)

The hierarchical nature of PIER input is depicted in ??. Each level is now briefly described.

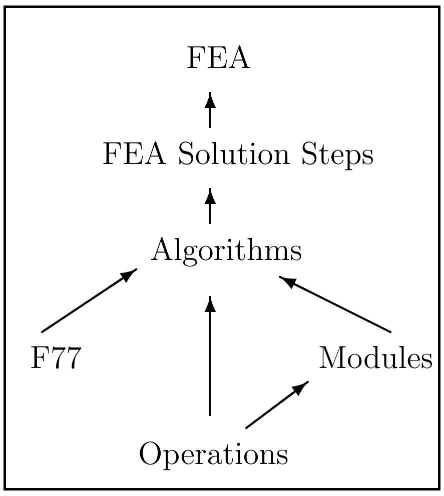

Figure 3: Hierarchy of Computations

- **Operation**: An operation is the smallest unit of computation specified by the user. 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

  $$z_0 = B^{-1}\, r_0$$

  $$\beta_2 = r_m \cdot z_m$$

  involved in the preconditioned conjugate gradient algorithm can each be specified by an operation (see page XXX). (Sharma: show actual spec for these equations somewhere and give reference here)

  An operation can specify either a symbolic or a numeric computation. For a statement, the variable on the left-hand side is its *output parameter*, while those on the right-hand side are its *input parameters.*

- **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.

- **Algorithm**: A numerical algorithm is specified by combining *modules* with `f77` constructs. PIER also supplies, from it knowledge base, certain standard algorithms that can be used directly.

- **FEA Solution Step**: A set of such algorithm specifications represents a FEA solution step. (Sharma: this is vague, name an example step, also how does one combine algorithms into a step?)

The PIER knowledge base contians a rich set of pre-defined operations and a number of frequently used numerical algorithms for FEA. A subset of operations defined in BLAS [] along with operations specific to the FEA application domain constitute the set of operations. Built-in operations include matrix-vector product, Crout element by element preconditioner, derivation of shape functions etc (Sharma: list many more).

To generate code for a key FEA solution step, the user must first define the describe the element mesh to PIER. The relevant informations i.e. element type, nodal coordinates and list of nodes associated with each element, are to be organized in an appropriate fashion for PIER consumption. This is followed by the definition of data objects (to be manipulated by FEA computations) and the input specifications for solution steps. PIER input specifications enable the user can specify

- element mesh;
- data objects storing element and nodal properties;
- a complete FEA procedure consisting of a sequence of FEA solution steps.

3 Describing the Element Mesh

To guide the derivation of FEA computations, PIER requires that user specify salient parameters of the (problem) domain and the type of the element used. It is assumed that the whole area of domain is covered by elements of same type. The syntax to specify domain parameters is

$$(\textbf{domain}\ \ \textbf{dim}{=}u_{dim}\ \textbf{nodes}{=}u_{nodes}\ \textbf{elements}{=}u_{elements}\ \textbf{name}{=}u_{name})$$

Where, keywords are boldfaced and other parameters are user supplied. The keywords *dim*, *nodes* and *elements* are used to specify geometric dimension of domain (u_{dim}), number of nodes (u_{dim}) and number of elements (u_{dim}) respectively.

To specify an element

$$(\textbf{element}\ \ \textbf{ldim}{=}u_{ldim}\ \textbf{nodes}{=}u_{nodes}\ \textbf{shape}{=}u_{shape}\ \textbf{name}{=}u_{name})$$

The keywords *ldim*, *nodes* and *shape* are used to specify the local nodal dimensions (u_{ldim}), number of nodes in the elements (u_{nodes}) and shape of the elements (u_{shape}) respectively. The keyword *name* in the syntax is used to assign a name (u_{name}) to an instance of domain or element.

To experiment with alternative domain discretization and/or element approximations, user may desire to define the problem domain and the discretizing element more than once. To specify the domain-element pair one uses,

$$(\textbf{use}-\textbf{domain}\ \ u_{name})$$

and

$$(\textbf{use}-\textbf{element}\ \ u_{name})$$

This immediately precedes specification for deriving FEA computations.

4 Defining Data Storage Schemes

Numerical algorithms employed in FEA solution steps manipulate well structured data objects (such as matrix). Most of the data objects are large one-dimensional or multi-dimensional arrays. Various storage schemes are used in practice to store data elements compactly. Also, the organization of data elements constituting a data object is domain and problem formulation dependent. Programming with data objects not stored in a conventional manner requires appropriately mapping references to data elements.

We address this issue by allowing the user to specify at the time of data object definition a desired storage style and, if appropriate, a domain specific name. In PIER, the syntax to specify a *dataobject* is

$$(\textbf{dataobject}\ \ \textbf{id}{=}u_{id}\ \textbf{type}{=}u_{type}\ \textbf{domain}{=}u_{do}\ \textbf{name}{=}u_{name}\ \textbf{storage}{=}u_{st})$$

The keyword *id* is used to specify an identifier which refers to the data object. It should be a valid F77 identifier. Keywords *type* represents the associated data type whereas *size* expects a list of integer values defining the array dimensions (for declaring domain specific data objects and scalar data objects this keyword is left unused). Keywords *domain* and *name* are used to specify symbolic names for the application domain (like FEA) and domain specific array name. Examples of few *name*s in FEA application domain are *esm* (element stiffness matrix), *gv* (global vector) etc. Desired storage styles are specified using *storage*, it expects a list of one or more than one valid storage styles. If no value is specified the conventional storage is assumed. Following is an example of complete syntax for specifying a global stiffness matrix a, stored in element-by-element fashion and each element matrix is symmetric.

$$(\textbf{dataobject}\ \ \textbf{id}{=}a\ \textbf{domain}{=}fea\ \textbf{name}{=}gsm\ \textbf{storage}{=}(ebye\ \ symm))$$

5 Defining PIER Operations

PIER operations can be viewed as a functional equation with one variable on the left hand side and an expression involving one or more than one variable on the right hand side.

$$v = F(u_1, u_2\ \cdots\ u_n)$$

In general the specification for a procedural abstraction like PIER operation has two parts: the interface specification and the behavioral specification. As indicated earlier, the behavioral specifications of PIER operations are part of the programming knowledge base and the user need only concern with the interface specifications.

To instantiate a PIER operation input variables names $u_1,\ u_2,\ \cdots\ u_n$ must refer to predefined data objects. The declaration for output data object v is automatically handled. In addition to input/output data objects, PIER operations require specifications for domain specific operation *execution style* and storage styles of input data objects.

The syntax for specifying a PIER operation is

$$(oprcode\ invar_1{=}u_{in_1}\ \cdots\ invar_n{=}u_{in_n}\ \textbf{outvar}{=}u_{L_{out}}\ \textbf{style}{=}u_{style}\ \textbf{tasks}{=}u_{tasks})$$

As usual, keywords are boldfaced. We now describe semantics of various keywords and parameters in the syntax.

- The PIER operation: The *oprcode* is the name of the desired operation. Few valid names are *mvp* (matrix-vector-product), *vip* (vector-inner-product) *assm* (assembly operator) etc.

- Input data objects : The keywords $invar_i$ *s* are used to specify names of the input data objects. The keywords for input data objects are different for each PIER operation. For example, operation *mvp* uses two keywords for input data objects namely *matrix* and *vector* whereas the keywords for *vip* are $vector1$ and $vector2$. u_{in_i} s are user supplied input data object names. Each name is appropriately matched with a keyword.

- Output data objects : The keyword *outvar* is used to represent the names of the user supplied output data objects ($u_{L_{out}}$).
- Style and Tasks : The keyword *style* can be used for user selected (domain specific) execution style (u_{style}). PIER provides the following styles for FEA domain

  1. *SC (scalar)* execute operation for one element at a time,
  2. *FP (fully parallel)* execute operation for all elements in parallel, and
  3. *BP (block parallel)* execute operation for a *block*[1] of elements at a time.

  The keyword *tasks* is used to specify number of processes or processors desired. In case of sequential processing, the keyword *tasks* is left unused and no value is given in place of u_{tasks}.

The example shows the syntax for instantiating *vip* operation with vectors $v1$ and $v2$. The output data are to be stored in s and the execution style is *bp* (block-parallel).

$$(vip \;\; vector1 = v_1 \; vector2 = v_2 \; \textbf{outvar} = s \; \textbf{style} = bp)$$

6 Specifying PIER Modules

As previously mentioned, a PIER module is a set of instantiated operations. All the operations in a module are without any control dependence and can be roughly viewed as a set of equations. The user constructs modules with instantiated PIER operations and specifies data object names at the entry and exit of the module. The module specifications are *operational* in nature as the set of constituting operation explicitly derive output data objects starting with input data objects.

A PIER module is specified as follows

$$(\textbf{module name} = u_n \; \textbf{invar} = u_{L_{in}} \; \textbf{outvar} = u_{L_{out}} \; \textbf{operations} = u_{L_{opr}} \; \textbf{composition} = u_c)$$

Where, $u_{L_{in}}$ is the list of input data objects into a module, $u_{L_{out}}$ is the list of output data objects from a module and u_n is the name of the module. The keyword *oprcode* is used to specify a list of instantiated operations, whereas u_c can have one of following two values.

1. *parallel* composition executes the code generated from more than one operation concurrently,

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

2. *sequential* composition executes code generated from each operation in a sequence.

For both composition styles, however, the numerical code produced by an operation may execute as more than one task.

7 Combining into Numerical Algorithms

After defining suitable modules using PIER operations, the control structure of a numerical algorithm is achieved using the F77 constructs. To use common names for data objects in F77 part and in PIER modules or operations is not allowed.

The user specifies following

$$(\mathbf{algorithm}\,\mathbf{step}{=}u_{step}\;invar_1{=}u_1\cdots\;invar_n{=}u_n\;\mathbf{outvar}{=}u_{outvar}\;\mathbf{method}{=}u_{method})$$

followed by the structure of algorithm. In generated code, the sequence of statements defined as modules are replaced in the algorithm structure by calls like

$$(\mathbf{pier\ module}{=}u_{module\text{–}name})$$

Where *step* refers to one of the FEA solution step and the keyword *method* is the algorithm to be used.

8 A Complete Example

Following is a complete PIER input specifications for preconditioned conjugate gradient solver for $A\,x = b$ system.

```
;Definition of module mod-a
(module name=mod-a invar=(a b) outvar=(x r p z) composition=parallel
        operations=((assign lhs=r rhs=b)(assign lhs=x rhs=0)
                    (assign lhs=p rhs=z)(croutebe invar=r outvar=z)))

;Definition of module mod-b
(module name=mod-b invar=(a x r z p) outvar=(t1 t3) composition=parallel
        operations=((vip vector1=r vector2=z outvar=t1)
                    (mvp matrix=a vector=p outvar=t2)
                    (vip vector1=p vector2=t2 outvar=t3)))

;Definition of module mod-c
(module name=mod-c invar=(alpha p x r t2) outvar=(x r t6)
        composition=parallel
        operations=((svp scalar=alpha vector=p outvar=t4)
                    (vva vector1=x vector2=t4 outvar=x)
                    (svp scalar=alpha vector=t2 outvar=t5)
                    (vva vector1=r vector2=t5 outvar=r)
                    (norm invar=R outvar=t6)))

;Definition of module mod-d
(module name=mod-d invar=(r) outvar=(t7) composition=parallel
        operations=((croutebe invar=r outvar=z)

;Definition of module mod-d
(module name=mod-e invar=(beta p) outvar=(p)
        operations=((svp scalar=beta vector=p outvar=t8)
                    (vva vector1=z vector2=t8 outvar=p)))

;Definition of main F77 program
(algorithm fem-step=linear-solver method=pcg-ebe matrix=a
           vector=b outvar=(x))
C       Main program
(pier module=mod-a)
 100    Loop from here
        (pier module=mod-b)
        alpha=t1/t3
        (pier module=mod-c)
        if (t6 .gt. 0.0001) then
        (piers module=mod-d)
        beta=t7/t1
        (piers module=mod-e)
        goto 100
        endif
        stop
        end
```

8.1 A Complete Example

We, now, give the complete specifications to generate subroutine to compute stiffness matrix. Here, we consider a 2D quadrilateral isoparametric element with 4 nodes. The problem domain is discretized into 1000 elements and the degree of freedom at each node is 2.

```
;Definition of Element Mesh
(domain dim=2 nodes=12 elements=1000 name=d1)
(element ldim=2 nodes=4 shape=quadrilateral name=e1)

;Deriving FEA Computations
(shapefunc element=e1 outvar=s method=isoparametric)
(bmatrix domain=d1 element=e1 outvar=b shapefunc=s method=displacement)

;Specifying input data objects - Assuming numerical values available
(dataobject id=x domain=FEA name=xnc storage=(ebye))
(dataobject id=y domain=FEA name=ync storage=(ebye))
(dataobject id=m domain=FEA name=mmx)

;Specifying ouput data objects
(dataobject id=k domain=FEA name=gsm storage=(ebye symm))

;Generate code for solution step 4
(stiff xnc=x ync=y mmx=m bmatrix=b jacobian=j
       outvar=(k) method=gauss-quadrature tasks=6)
```

9 Summary

To be written.

References

[1] Chang, T. Y., "NFAP - A Nonlinear Finite Element Program, Vol. 2 - Technical Report," College of Engineering, University of Akron, Akron, OH. 1980.

[2] "*COSMIC NASTRAN USER*'s manual," Computer services, University of Georgia, Columbus, GA. USA 1985.

[3] Fritzson, Peter and Fritzson Dag, "The Need for High-Level Programming Support in Scientific Computing Applied Mechanical Analysis",

[4] Petti, R., "The Role of Symbolic Mathematics Software in Mathematical Modeling", Winter Annual Meeting, ASME Nov. 27-Dec. 2 1988.

[5] Russo, Mark F., Peskin, Richard L. and Kowalaski, A. Daniel, "Using Symbolic Computation for Automatic Development of Numerical Programs", Coupling Symbolic and Numerical Computing in Expert Systems, II, Bellevue, Washington, July 20-22, 1987.

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

[7] Kant, E., Daube, F., MacGregor, W., and Wald, J., "Synthesis of Mathematical Modeling Programs", TR-90-6, Schlumberger Laboratory for Computer Science, Austin, TX 78720.

[8] Cook, Grant O., "ALPAL, a Program to Generate Simulation Codes from Natural Descriptions", UCRL-102076, Lawrence Livermore National Laboratory, L-35, Livermore, CA 94551.

[9] Sharma, N., "Generating Finite Element Programs for Warp Machine," Proceedings of ASME Winter Annual Meeting, Chicago, IL, Nov. 1988.

[10] Sharma, N., and Wang, Paul S., "Generating Parallel Finite Element Programs for Shared-Memory Multiprocessors," Proceedings of ASME Winter Annual Meeting, Dallas, TX, Nov. 1990.

[11] Sharma, N. and Wang Paul S., "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, Rome, Italy, July 4-8, 1988 pp. 33-56.

[12] Sharma N., "Generating Finite Element Programs for Multiprocessors", Fifth SIAM Conference on Parallel Processing for Scientific Computing, March 25-27, 1991, Houston, TX.

[13] Sharma, Naveen and Wang, P. S., "Guide to Parallel Fortran Programming on the Sequent Balance", Internal Report, Department of Mathematics and Computer Science, Kent State University, Kent, OH 44240, 1989.

[14] Sharma, N., "PIER: A Parallel Finite Element Analysis Programmer", Doctoral Proposal, Department of Mathematics and Computer Science, Kent State University, Kent, OH 44240. September 1991.

[15] Tan, Trevor, and Wang, P. S., "Automatic Generation of Parallel Code for the Warp Parallel Computer", Proceedings, 1st International Workshop on Computer Algebra and Parallelism, June 1988, Grenoble, France.

[16] Gross T., and Lam, M.,"A Brief Description of W2", Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[17] Weerawarana, Sanjiva and Wang, Paul S., “GENCRAY: User’s Manual,” Department of Mathematics and Computer Science, Kent State University, Kent, November 1988.

[18] “Balance Technical Summary,” Sequent Computer Systems Inc., Beverton, OR. 1988.

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

[20] Annaratone M., Arnould E., Gross T., Kung H. T., Lam M. S., Menzilcioglu, O. and Webb J.A., “The Warp Machine: Architecture, Implementation and Performance,” IEEE Transactions on Computers, Vol. C-36,No. 12, Dec. 1987, pp. 1523-1538.

[21] Babb, R. G., “Parallel Processing with Large Grain Data Flow Techniques”, IEEE Computer, July 1984.

[22] O’Hallaron, D. R., “The ASSIGN Parallel Program Generator”, CMU-CS-91-141, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 1991.

[23] Allen, J. R. and Kennedy, K., “PFC: a program to convert Fortran to parallel form”, Supercomputers: Design and Applications, K. Hwang, editor, IEEE Computer Society Press (1985), pp 186-205.

[24] Kuck, D. J., “The Structure of Computers and Computations”, Volume 1, John Wiley and Sons, New York, 1978.