Generating Finite Element Programs for Shared Memory Multiprocessors

Naveen Sharma and Paul S. Wang [1]
Department of Mathematics and Computer Science
Kent State University
Kent, OH 44240-0001

ABSTRACT
Symbolic computation is employed to automatically derive formulas in finite element analysis (FEA) and to generate parallel numeric code. Key FEA computations parallelized include element stiffness computations and solution of global system of equations. An element-by-element preconditioned conjugate gradient method is used to solve the global system of equations in parallel. Derived formulas are automatically mapped onto the shared-memory architecture. An experimental software system, P-FINGER, is being extented. P-FINGER features a specification language to describe numeric algorithms for which code is to be generated. The specifications also allow an automatic code dependence analysis mechanism to extract parallelism from the specified computational steps. A separate code translator GENCRAY is modified to render code into parallel f77. Generated parallel routines run under the control of existing FEA packages. Examples of generated code are also presented.

[1]Work reported herein has been supported in part by the National Science Foundation under Grants CCR-8714836 and EET-8714628

1. INTRODUCTION AND BACKGROUND

Finite element analysis (FEA) is a technique for engineering analysis of complex linear and nonlinear problems. FEA is frequently used for solving boundary and initial value problems that arise in stress analysis, heat transfer and continuum problems of all kinds. FEA computations are CPU intensive and involve large data sets. Practical FEA problems, especially in higher dimensions, are very time consuming on regular sequential computers. The advent of affordable parallel computers provides a chance to perform FEA with reasonable speed and cost. Large numerical packages such as NFAP(Refs. 1) and NASTRAN(Refs. 2) exist for FEA applications. FEA packages, we know of, are all developed for sequential computers and they provide facilities for frequently used models and cases. Only slight modifications of the "canned" computational approaches are allowed via parameter setting. Without substantial change, these packages can not deal with new formulations, materials, solution procedures, or parallelism.

The main calculations in a finite element analysis (FEA) program involve computing element contributions, assembling the system from these contributions and solving the resulting system. We identify *key* phases of FEA which are *compute intensive* and are *reprogrammed* every time a new formulation is used. Our approach is to employ symbolic computation to automatically generate sequential and parallel codes for key phases of FEA. Based on user input, quantities such as the strain-displacement matrices, the element stiffness matrices and the material property matrices can be derived using symbolic mathematical computations. Resulting formulas are used to generate code for sequential computers, the Warp systolic array computer, and shared memory multiprocessors (SMP) such as the Encore Multimax and the Sequent Balance. Generated code can be executed in conjunction with existing f77-based FEA packages. This approach relieves the engineer of tedious symbolic derivations and automates the coding and parallelization of the numeric programs required.

We have been working in the above described direction for a number of years (Refs. 3,4,6). Reported here is our most recent work on parallelizing FEA computations on the popular SMP's. At Kent we have in our department two SMP's: a 12 processor Encore Multimax [2] and a 26 CPU Sequent Balance making experimentation quite easy.

We selected the element stiffness matrix generation and the solution of global system of equations as key parts of FEA to parallelize. One of the most compute-intensive phases of FEA is the solutions of the global system of equations. The element matrices are dense, but the resulting system matrix is sparse due to local nature of the method. Direct solution meth-

[2]NSF CISE Research Instrumentation Grant CDA-8820390

ods based on Gaussian elimination have been used extensively. The direct methods require assembly of the global stiffness matrix and become less appealing for large systems. Recently iterative methods have been applied with encouraging results. Element-by-element (EBE) schemes used in conjunction with an iterative method does not require assembly of the global matrix and involves computations strictly at the element level. Specifically, our work concentrates on the preconditioned conjugate algorithm using EBE technique. The PCG/EBE method is suitable for parallelization and is the one we adopted for our code generation research.

To make our numeric code generation scheme flexible and easy to adopt for other algorithms and solution methods, we have developed a numeric algorithm specification language for use within P-FINGER (Fig. 1). The specification allows a programmer to list *blocks* of computational statements and attach properties to each statement forming a structure called a *flownode*. The flownodes are used in code dependence analysis and in derivation of numeric code for the target architecture. Actual code for the Sequent Balance is generated with a modified GENCRAY (Refs. 7). Generated code is then compiled and linked with an existing f77 based finite element package and overall speed up can then be measured.

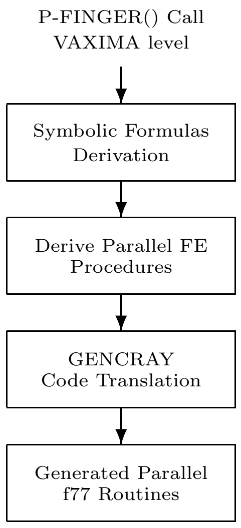
P-FINGER() Call
VAXIMA level
Symbolic Formulas Derivation
Derive Parallel FE Procedures
GENCRAY Code Translation
Generated Parallel f77 Routines

Figure 1: Parallel FE Program Generation

The GENCRAY code generator

P-FINGER runs under the Macsyma system to perform the symbolic computations necessary to derive finite element code. Once the lisp internal representations of the code has been derived and properly mapped onto the target architecture, a separate *code translator* package is used to translate them into correct numerical code in a target language.

The code translator used is GENCRAY which has been developed by our group at Kent originally to generate code in Cray f77. The GENCRAY has been extended for the purposes of this investigation to also generate parallel f77 code for the Sequent Balance.

2. SHARED MEMORY MULTIPROCESORS

A major part of the present research has been conducted on the Sequent Balance which is a shared memory multiprocessor manufactured by Sequent Computer Corp. It has 26 tightly-coupled processors, each a 32-bit National Semiconductor processor capable of executing 0.75 MIPS. The main memory of 32 MB is shared by all processors.

The Balance (Refs. 8) runs under the DYNIX operating system which is Sequent's UNIX system. In addition to all the usual UNIX facilities, DYNIX provides capability of multiprocessing, employing all the CPUs available to support concurrent processes by running up to 26 of them simultaneously. The processors are shared by all the processes including operating system and user processes. The f77 and C languages are extended with parallel features and augmented by parallel library routines. It is possible to reserve a certain number of processors to be dedicated to a certain parallel program for the duration of its execution. This makes performing parallel experiments easy. Programming primitives for allocating shared memory, synchronization and timing of parallel processes are provided. An interactive debugger dealing with multiple processes also exists.

Parallel activities are initiated in a program by creating *child processes* or *tasks* that execute independently but simultaneously with the *parent process.* Each task is essentially a separately running program. Communication among a group of cooperating parallel tasks is achieved through *shared memory.* Data stored in shared memory by one task are accessible by all other tasks sharing the same (virtual) memory area. A program can generate many child tasks. But if they are going to be executed in parallel, the total number of tasks is limited to the total available CPU's.

3. SYMBOLIC DERIVATION OF FINITE ELEMENT FORMULATIONS

Shape Function Derivation

P-FINGER is capable of deriving element shape functions, strain-displacement matrices and element stiffness formulations for isoparametric and plane triangular elements. This work has been reported in (Refs. 16,18). Defining a suitable shape function is very importance in FEA. Choice of shape function can significantly affect finite element approximation. Recent research activities (Refs. 9,10) testify to the usefulness of symbolically derivating shape functions for many types of elements.

A large number of different techniques are used for constructing shape functions. We think it is neither feasible nor desirable to treat all of them automatically in one package. With this view, we are working on a general framework within which a finite element engineer can guide the derivation of any desired shape functions. The framework will support shape function derivation in the following steps:

User Input

- *Domain information*
  1. Type of the problem to be solved e.g. 2D plane problem or 3D axisymmetric problem etc.
- *Element Information*
  1. Geometric shape
  2. Number of nodes
  3. Type of nodes (exterior or interior)
  4. Type of nodal variable (degrees of freedom at each node)
  5. Element type (C^0 type or C^1)
- *Interpolation Approximation function*
  1. Specify coordinate information (i.e. choice of global coordinates or local coordinate system).
  2. Define approximating polynomials (if necessary).

Derivation

P-FINGER provides a fully automated procedure for constructing polynomial-based shape function, including isoparametric formulations. Other shape functions can be derived interactively by using primitives provided in P-FINGER. Primitives supplied by package are expressions for:

1. *Lagrange polynomial* $L_k(x)$ of a given degree n

   $$\frac{x-x_0}{x_k-x_0}\cdot\cdot\frac{x-x_{k-1}}{x_k-x_{k-1}}\cdot\frac{x-x_{k+1}}{x_k-x_{k+1}}\cdot\cdot\frac{x-x_n}{x_k-x_n}$$

   on specifying the value of field variable at nodes shape functions can be written at once using $L_k(x)$ of proper degree.

2. *Hermite polynomial* for degree n

   $$H_{mi}{}^{n}$$

   where, m and i are order of derivative and element node respectively. These polynomials are useful in constructing shape functions when field variable, as well as its derivative, is specified at nodes.

3. *Jacobian matrix* J for coordinate transformations. Derived shape functions are used to construct this matrix.

We continue to add more primitives for easy derivation of many other types of shape functions.

Testing
All derived interpolation functions are tested for *compatibility*, *completeness* and *geometric isotropy* [16].

4. ELEMENT-BY-ELEMENT SOLUTION SCHEME

Recent years have seen the development of algorithms for solving equations without actually assembling the global stiffness matrix (Refs. 11,19) which is often very large. The algorithm is based on the element-by-element (EBE) solution scheme. The idea behind EBE can be exploited with any of the residual-based iterative methods (Refs. 17). The main advantages of EBE solvers (Ref. 12) are significant savings in numerical operations, computer storage and potential for parallel implementation. In our work we have elected to use the preconditioned conjugate gradient (PCG) method (Refs. 20) to accelerate the basic iterative method.

Let us briefly outline the PCG algorithm for the solution of a linear system

$$A\,x = b$$

Given a preconditioner B and an initial approximation x_0 the algorithm produces a series of improved approximations x_m to which corresponds a sequence of diminishing residuals $r_m = b - A\,x_m$.

Algorithm PCG
Input : $A\,x = b$, x_0, B
Output: x for which residual $r = b - A\,x$ is acceptable.

- **PCG1.** Initialize loop counter m and solution vector x.

$$m = 0$$

$$x_m = \mathbf{0}$$

Initializing residual r_m and vector z_m.

$$r_m = b$$

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

Initializing conjugate search direction p.

$$p_m = z_m$$

- **PCG2.** Line search to update solution and residual.

$$alpha_1 = r_m \cdot z_m$$

$$alpha_2 = p_m \cdot A\,p_m$$

$$\alpha^m = \frac{alpha_1}{alpha_2}$$

Updating solution vector x and residual.

$$x_{m+1} = x_m + \alpha^m\,p_m$$

$$r_{m+1} = r_m - \alpha^m\,A\,p_m$$

- **PCG3.** If convergence criterion for r is satisfied then return.
- **PCG4.** Compute new conjugate direction.

$$z_{m+1} = B^{-1}\,r_{m+1}$$

Line search to update conjugate direction p.

$$beta_1 = r_{m+1} \cdot z_{m+1}$$

$$beta_2 = r_m \cdot z_m$$

$$\beta^m = \frac{beta_1}{beta_2}$$
$$p_{m+1} = z_m + \beta^m \, p_m$$
$$m = m + 1$$

Go to step PCG2.

[XXX We used two different preconditioners:

1. *Crout element-by-element preconditioner*: a preconditioner defined using element level factorizations (Refs. 11)
2. *Jacobi preconditioner*: a preconditioner using the main diagonal of the assembled global matrix.

For the latter, the diagonal inverse can be partitioned easily for EBE computations.
In the PCG algorithm, most of the operations involve matrix-vector or vector-vector products. The EBE idea can be exploited by keeping one of the operands at the element level and assembling the other. Results of the EBE operations are accumulated appropriately. Details can be found in (Refs. 17,19).
XXX]
A major goal of our research is to study parallel implementations of the PCG/EBE method on shared memory multiprocessors and to generate parallel PCG/EBE code automatically.

5. FEA ON THE SHARED MEMORY MULTIPROCESSORS

Our strategy for FEA on shared memory multiprocessors is to identify key parts in the FEA computation and generate parallel code for them. The parallel routines executes under the control of an existing finite element analysis package, NFAP, which runs as a sequential program on the same shared-memory multiprocessor. The effectiveness of the parallel strategy is then measured by the speed up over the sequential version of the same algorithm.

The most computation intensive parts in FEA that we choose to parallelize are

1. Computing the element strain-displacement matrix
2. Computing the element stiffness matrix
3. Solution of the global system of equations

Our prior work on items 1 and 2 on the Warp gives us good experience to parallelize them again on the SMP and to extend the scope the parallel computations beyond the element computation level to include item 3.

There are at least two alternative ways to parallelize item 3. One method is to form the global stiffness matrix then to use a standard parallel elimination algorithm. The other approach is to use an iterative EBE algorithm that produces the solution without ever assembling the global stiffness matrix. We choose to experiment with generating parallel code for the PCG/EBE algorithm.

We have devised parallel schemes on the SMP for all three key items mentioned above. Furthermore, we appropriately interface the parallel element stiffness computation to the parallel equation solving phase. Fig. 2 gives an overview of NFAP interfacing with generated parallel routines. We shall discuss details of our parallel FEA steps in the next section.

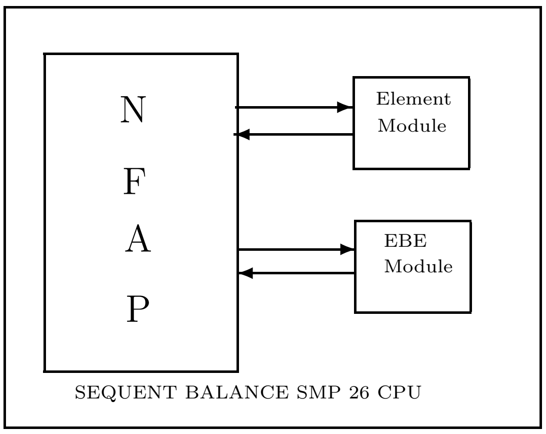

Figure 2: Program Interfaces

6. PARALLEL FE COMPUTATIONS

We describe parallelization of key FEA computations in this section. Our target architecture is Sequent Balance. The parallel programming environment on Balance supports *multitasking*. A single application, using multitasking, can consist of multiple processes executing concurrently. Parallel programming library provides multitasking extensions. P-FINGER generates parallel routines for key computations of FEA in Sequent parallel f77.

Parallel Element Computations

Each element in the FE domain is defined by such data as nodal coordinates, element thickness and material properties. Given information about element geometry and problem domain, P-FINGER is capable of deriving element formulations. Computation of strain-displacement matrix involves evaluation of derived formulas. However, for computing element stiffness, integrals of a rational function are needed. Generated code is used to compute the integrals using Gaussian quadrature. In case of uniform finite element mesh, identical element computations are repeated for all elements and can be carried out simultaneously by multiple processes on an SMP.

We use a *multitasking* model, based upon *data partitioning*, to parallelize the element computations. In multitasking, a number of processes with the same program can be set up to compute over different elements in parallel. The element data and the results are stored in shared memory. Each process accesses the shared memory for the required data and stores results back into shared memory areas. The entire element description data are divided among the parallel processes. This parallel scheme simulates a SIMD environment. Fig. 3 pictorially depicts the idea. There are 20 processes and each process P_i execute identical program for element computations. All tasks must finish before we go to the next phase of our parallel computation.

Multitasking experiments have already been performed on the Warp systolic array computer (Refs. 3,6). This provides a performance level against which the effectiveness of the shared memory multiprocessors for data partitioning parallelism can be compared.

Parallel PCG/EBE

P-FINGER exploits parallelism in PCG/EBE algorithm at the level of each numerical operation in the algorithm. The operations involved in steps PCG1, PCG2 and PCG4 are of type matrix-vector multiplications, inner products of vectors, scalar-vector products etc. Each operation is carried out for all elements and nodes in the problem domain. Our strategy is to parallelize each such array/vector operation in the PCG/EBE algorithm.

On an SMP, parallel array/vector operations are achieved through scheduling loops over the available number of processes. On the Sequent Balance, loops can be scheduled *statically* or *dynamically*. Static scheduling is attractive because it involves minimal run-time overhead. For static scheduling to be feasible the following conditions must be met (Refs. 14):

1. There are no branches in the loop body, i.e. amount of computations within one loop body is known at compile time.

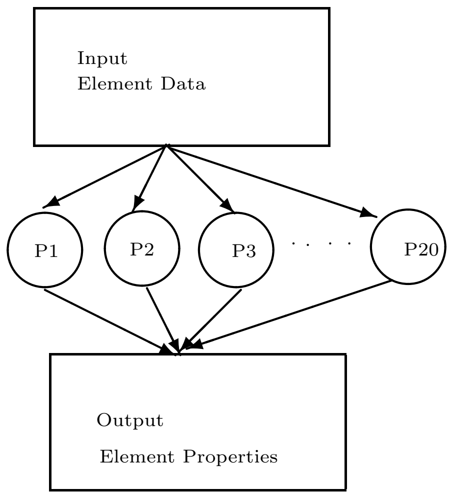

Figure 3: DATA PARTITIONING MODEL FOR ELEMENT COMPUTATIONS

2. The number of loop iterations is known before the loop begins to execute.

Steps in the PCG/EBE algorithm satisfy the above conditions enabling us to schedule loops statically.

One important performance consideration on the SMP is the shared access of data and the storing of results back into a structure shared by parallel tasks. If the same data is accessed simultaneously by two different tasks, they have to avoid a time collision and will slow each other down. We have taken care to partition the elements into largely disjoint groups. A group contains elements which do not share nodes with on another allowing them to be processed simultaneously. The different groups are executed sequentially.

We parallelize PCG/EBE algorithm using one of the two different multitasking approaches. In addition to data partitioning, Sequent parallel programming environment provides for *function partitioning* also.

[XXX **Data Partitioning for PCG/EBE**

The data partitioning technique discussed in the previous section can also be applied here for algorithm PCG. For a totally parallel operation in PCG, tasks executing the same code would work on different sets of element data. For a group parallel numeric operation, P-FINGER splits the elements in disjoint groups. Numeric operation for each group of elements is carried out with data partitioning. The different groups are, however, executed sequentially.
An alternative to data partitioning is function partitioning which is our next topic. XXX]

Function Partitioning for PCG/EBE

[XXX We are interested in tasks that execute different numerical operations of the PCG/EBE algorithm in parallel. The order of execution for the numeric operations is constrained by their data dependence. Operation A is dependent on operation B if A uses the result produced by B. We have developed and implemented an algorithm in P-FINGER, to carry out the necessary data dependence analysis for a block of numeric operations in the PCG/EBE algorithm.
The algorithm produces a layered dependence graph. Each layer contains numerical operations with mutual data independence. *All operations within the same layer can be executed in parallel.* For example, in block PCG2, the quantities $alpha_1$ and $alpha_2$ can be computed in parallel. Later in the same block, vectors x and r are updated simultaneously.
XXX] Within a layer, each operation can be allocated a number of processes and further parallelized with data partitioning. Disallowing simultaneous executions of data dependent operations eliminates the need for explicit synchronization.
Obtaining an optimal schedule for executing a set of dependent tasks on a given number of processors is a hard problem (Refs. 15). We are working on heuristics for statically scheduling a layered dependency graph.

The element computations and equation solution are two parallelized phases of the PGC/EBE algorithm. It is important to interface them correctly for better efficiency. Fig. 4 outlines our parallel finite element scheme.
XXX]

7. NUMERICAL CODE GENERATION

Based on problem domain parameters and formulae derived, P-FINGER proceeds to generate numeric code in modules that can readily interface with existing software packages. Current capabilities include - generating f77 based sequential code and parallel code for Warp systolic array computer and for the Sequent Balance SMP.

We will focus on the automatic generation of parallel code for the SMP. The code generation phase can be divided into four stages as shown in Fig 5. A description of each stage will be given.

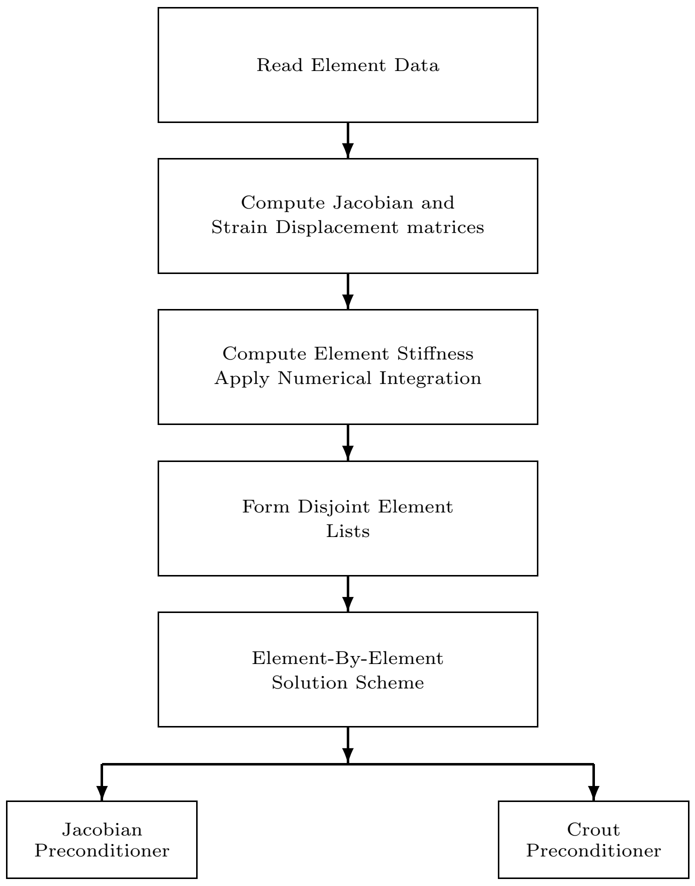
Read Element Data
Compute Jacobian and Strain Displacement matrices
Compute Element Stiffness Apply Numerical Integration
Form Disjoint Element Lists
Element-By-Element Solution Scheme
Jacobian Preconditioner
Crout Preconditioner

Figure 4: Overview of parallel finite element scheme

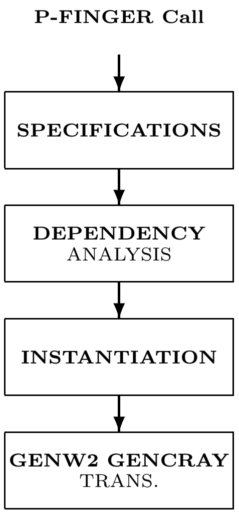

Figure 5: Overview of Code Generation

Specification of Numerical Algorithm

In order to make the numeric code generation procedure flexible, we have developed a specification language to describe the numeric algorithm for which code is to be generated. The Gaussian quadrature and PCG algorithms have been specified in P-FINGER this way.

A *statement* in the algorithm is represented as an equation with one variable on the left hand side and an expression involving one or more variables on the right hand side. Some statements in the PCG algorithm (section 4) are

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

$$beta_2 = r_m \cdot z_m$$

Step PCG3, which tests for iteration condition, is not a statement. Such fixed programming controls are included as part of the program generation template.

The numerical algorithm is divided in *blocks* (Refs. 13) so that each block contains a sequence of statements with no entry or exit points except the beginning and end of the block.

In other words, control flow enters at the beginning and leaves at the end of a block. The blocks must be specified in the program. Examples blocks in the PCG procedure are PCG1, PCG2 and PCG4.

Each block is characterized by IN and OUT sets. All variable names appearing on left-hand sides of equations form the OUT set. Variables in the expressions on the left-hand sides give rise to the IN set. The problem-dependent blocks fits into a program control structure to form a function or subroutine. Such control structures are represented by *templates*.

Each equation or statement in the block also has an attached set of attributes.

1. *oprcode*: The high level operation carried out by the statement: matrix multiplication, inner products, integration, etc.
2. *inset*: The set of all right-hand side variables in the statement.
3. *outset*: The one variable on the left-hand side of the statement.
4. *execution style*: One of the followings:
   - Scalar
   - Fully parallel
   - Group parallel The meaning of these styles will be described in the next subsection.

A statement/equation together with these attributes is a structure called a *flownode*. The flownodes help generating efficient code and mapping algorithms onto parallel computers.

In addition to the above specifications, each variable is also declared with

(*type var dim1 dim2 ...*)

So scalar, vector and matrix/array quantities are handled correctly and efficiently.

[XXX **Dependency Analysis**
Let us now focus on generating parallel code based on the above specifications. The parallelism for operations within a block can be automatically derived from information contained in the flownodes. An algorithm has been developed to analyze flownodes in a given block for mutual data dependency. The dependence is determined by examining the inset and outset for each flownode.

The algorithm computes a *layered dependence graph*(LDG) for the block. Statements grouped in same layer have no mutual data dependencies and can be executed in parallel.

Data Dependence Algorithm

Details of our data dependence algorithm is presented here. The algorithm takes L, the list of flownodes in any given block, as input and returns R, a layered dependence graph of the block. The algorithm begins by collecting all flownodes whose insets have been computed already in a list U. The same flownodes are deleted from L. U forms the first layer in the dependence graph and is added to R.

Let us now describe constructing the next layer. The algorithm compares the inset of each flownode left on L with the combined OUT set of all existing layers on R. A flownode is included in the next layer and deleted from L if its inset is a subset of the combined OUT set. The process continues until L is empty.
XXX]
Input: L, the set of flownodes in a block.
Output: R, a layered graph representation, initially an empty list.

```
1. Collect, in a set U, all the nodes with an empty inset or with an inset of constants and/or
   variables belonging to OUT sets of previous blocks. (Note the OUT set of a block is not
   the outset of a flownode.)

2. R = NULL

3. while (L is not empty) do steps (4) and (5)

4. /* set difference */
   L = L − U
   Add list U on the list R.
   /* number of elements in L */
   N1=LENGTH(L)

5. for i = 1 to N1 do
       N2 = number of subsets in R
       /*Flag to count the number of variables matched*/
       F = 0
       for j = 1 to N2 do
           N3 = LENGTH($R_j$)
           for k = 1 to N3 do
               /* Comparing inset of flownode with oustset of LDG layer */
               if(inset ($L_i$) $\bigcap$ outset($R_j$(k)) != NULL) then
```

```
                Construct a data link between $L_i$ and $R_j$(k)
                F = F + 1
            endif
        endfor[k]
    endfor[j]
/* If all data requirements of flownodes are met */
if (F = LENGTH(inset($L_i$))) then
    Add $L_i$ on to list U
endif
endfor[i]
/* If newly formed layer is NULL */
if (U = NULL) then
    Add L on to list R
L = NULL
endif
```

6. return(R)

The result R is a list of sublists, each sublist contains flownodes with no mutual data dependence. The property lists associated with each flownode stores data links for the flownode.

Instantiation

A piece of numeric code is generated for a block by *instantiating* all its flownodes. The instantiation of each type of oprcode is supported by a separate function in P-FINGER. An instantiating function takes inset and outset as arguments and returns a piece of code implementing the operations. The instantiation process is summarized as follows.

- Extract oprcode, inset and outset from a flownode
- Call instantiating function *oprcode(inset, outset)*
- Collect variable names and dimensions for data declaration

The numeric codes thus derived are in internal LISP representation.

Code Translation

Instantiated code for each flownode is used to construct the code for an entire block. LDG is used to order flownodes so that the data dependence requirements are satisfied. Finally, GENCRAY is called to translate the LISP representation into f77 code. Templates are used and code for each block is placed appropriately.

Generating Parallel Code for Flownode

As we have mentioned earlier, a flownode in the PCG/EBE belongs to one of three classes: scalar, fully parallel and group parallel. We call these classes *execution styles*. Information about execution styles of an operation is given to P-FINGER at algorithm specification time.

We now describe generating parallel code for a flownode specified as fully or group parallel. Steps are summarized in Fig. 5.

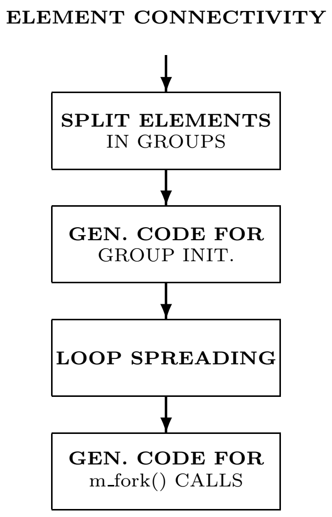

Figure 5: GENERATING CODE FOR PARALLEL OPERATIONS

1. Split finite elements in disjoint groups. An element blocking algorithm suggested by Hughes et al (Refs. 11) is implemented. The algorithm proceeds after given information on the connectivity of the finite element mesh. The input consists of the number of elements to be divided in disjoint groups and the maximum size of the groups. Disjoint groups are created to partition the given elements.

2. Generate assignments of elements to group names.

3. Generate a subroutine to implement the flownode operation.

4. Generate multi-tasking code to execute multiple copies of the subroutine to work on disjoint groups in parallel. For a fully parallel operation, all elements form one group and all elements are processed simultaneously. Disjoint groups are formed for group parallel operations using a blocking algorithm. In this case all elements within a group are processed at once. Groups are executed one by one.

In the next section examples of generated codes for the fully parallel and group parallel cases will be given.

8. EXAMPLES OF GENERATED CODE

Fully Parallel Code

In the PCG algorithm, the block PCG4, contains five statements. An example of generated code is presented here. The code is generated for a finite element problem with the following characteristics.

- The domain is discretized into 256 triangular elements.
- There are 153 nodes and each node has 1 local degree of freedom.
- The Jacobian preconditioner is used.

As can be seen from the code, all global vectors are manipulated at the element level. Number of tasks used is 10.

```
c**** begin generated code ****
C  Block Begins Here
       . . . . . . . . . .
       . . . . . . . . . .
C    Computing Beta coefficient
      do 1034 l2582=1,256
         do 1035 i2583=1,3
            i36=gnode(l2582,i2583)
            do 1036 i2584=1,1
               i37=1+(-i2584)
               i38=i37+(1*i2583)
               i39=i37+(1*i36)
               z(i39)=zarray(l2582,i38)
 1036       continue
 1035    continue
 1034 continue
```

```
      if (m_fork(bnum,26,25,6) .ne. 0) then
         stop
         end
      endif
      subroutine bnum (size1,size2,switch)
      integer size1,size2,switch
      common /block/r2(256,3),z(153),bnum
      real t43
      integer size,e1,e2,pid,s
pid = m_get_myid()
      if (pid .le. switch) then
         size=size1
         e1=(pid*size)+1
         e2=(pid+1)*size
      else
         size=size2
         s=(switch+1)*size1
         pid=pid-(switch+1)
         e1=s+(1+(pid*size2))
         e2=s+((pid+1)*size2)
      endif
      do 1037 l2585=e1,e2
         do 1038 i2586=1,3
            i40=gnode(l2585,i2586)
            do 1039 j2587=1,1
               i41=j2587+(-1)
               i42=(1*i40)+(-i41)
               t43=t43+(r2(l2585,i42)*z(i41))
 1039       continue
 1038    continue
 1037 continue
      call m_lock
      bnum=bnum+t43
      call m_unlock
      return
      end
      beta=bnum/anum
```

```
C     Updating p vector
      if (m_fork(pup,16,15,3) .ne. 0) then
         stop
         end
      endif
      subroutine pup (size1,size2,switch)
      integer size1,size2,switch
      common /block/z(153),beta,p(153),ptemp(153)
      integer size,e1,e2,pid,s
pid = m_get_myid()
      if (pid .le. switch) then
         size=size1
         e1=(pid*size)+1
         e2=(pid+1)*size
      else
         size=size2
         s=(switch+1)*size1
         pid=pid-(switch+1)
         e1=s+(1+(pid*size2))
         e2=s+((pid+1)*size2)
      endif
      do 1040 m2588=e1,e2
         z(m2588)=z(m2588)+(beta*p(m2588))
 1040 continue
      return
      end
      go to 657
c *** end generated code ***
```

Group Parallel Code

We present an example of the generated code for matrix-vector product which is a group parallel operation in PCG2.

$$ap = A\,p_m$$

which is part of computing $alpha_2$. The finite element problem has the same characteristics as in the previous example. The number of tasks used is 10.

```
c *** begin generated code ***
      . . . . . . . . . . 
      real karray(256,6)
      real pg(153)
      real apvec(153)
      integer b16(44)
      . . . . . . . .
      . . . . . . . .
      integer b10(44)
      integer b9(44)

C     Assigning elements to blocks

      b16(1)=250
      b16(2)=247
      b16(3)=244
      . . . . . .
      . . . . . .

      b12(35)=26
      b12(36)=23
      b12(37)=20
      b12(38)=17
      b11(1)=238
      b11(2)=235
      b11(3)=232
      . . . . .
      . . . . .

      b9(15)=7
      b9(16)=4
      b9(17)=1
      . . . . . . . . . . . . . . . . . . . .
C     Calling subroutine matvec for block b16
      if (m_fork(matvec,b16,1,1,0) .ne. 0) then
         stop
      endif
      . . . . . . . . . . . . . . . . . . . .
C     Calling subroutine matvec for block b12
```

```
      if (m_fork(matvec,b12,4,3,8) .ne. 0) then
         stop
      endif
C     Calling subroutine matvec for block b11
      if (m_fork(matvec,b11,4,3,7) .ne. 0) then
         stop
      endif
C     Calling subroutine matvec for block b10
      if (m_fork(matvec,b10,4,3,3) .ne. 0) then
         stop
      endif
C     Calling subroutine matvec for block b9
      if (m_fork(matvec,b9,2,1,7) .ne. 0) then
         stop
      endif
      . . . . . . . . . . . . . . . . .
      . . . . . . . . . . . . . . . . .

      subroutine matvec (bname,size1,size2,pidnum)
      real bname(44)
      integer size1,size2,pidnum
      integer pid
      common /blk/karray,pg,ap
      pid = m_get_myid()
      if (pid .lt. pidnum) then
         isize=size1
      else
         isize=size2
      endif
      do 1001 i2553=((pid*isize)+1),((pid+1)*isize)
         i2554=bname(i2553)
         do 1002 k2555=1,3
            do 1003 k2556=1,3
               i17=(k2555+(-1))/2
               i18=(5*i17)+k2556
               i19=gnode(i2554,k2556)
               apvec(i19)=apvec(i19)+(pg(i19)*karray(i2554,i18))
```

```
 1003       continue
 1002    continue
 1001 continue
      return
      end
c *** end generated code ***
```

9. SUMMARY

Using a symbolic computation system to generate parallel numerical code is a practical way of combining powers of symbolic systems and parallel machines. An experimental software system P-FINGER is being implemented to generate parallel programs for key finite element phases. We described an effective way of parallelizing FEA programs on shared memory multiprocessors. We have extented P-FINGER not only to generate code for the Sequent Balance but also to generate parallel code for the equation solution phase of FEA using the PCG/EBE method. The key elements of generating FEA programs are:

1. Symbolic derivation of finite element formulas and code
2. Mapping of computations onto parallel architecture
3. Generating appropriate high level code to run with an existing FEA package

To make code generation flexible, we have developed a numeric algorithm specification language to use within P-FINGER. This language allows blocks of statements to be described with attached properties forming *flownodes*. The flownodes enables us to derive and perform dependence analysis for parallelism.

Also described are two parallel models for mapping PCG/EBE onto the Sequent Balance. Examples of generated code are presented. We are conducting timing experiments with the generated code. Timing results will be presented at the conference.

10. ACKNOWLEDGEMENT

We would like to thank Professors T. Y. Chang, N. Ida and A. Saleeb of Univesity of Akron for actively participating in several discussions over the course of this research work. Professor Chang and his research group also provided access and help with the NFAP package. Mr. S. Weerawarana implemented GENCRAY and assisted with interfacing it to P-FINGER.

11. REFERENCES

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

(2) *COSMIC NASTRAN USER*'s manual. Computer services, University of Georgia, USA.1985.
(3) Sharma, N., "Generating Finite Element Programs for Warp Machine", appeared at ASME Winter Annual Meeting, Chicago, IL, Nov.1988.
(4) Sharma, N., Chang, Paul T., Wang, Paul S., "Generating Parallel Finite Element Programs for Shared-Memory Multiprocessors", Unpublished document, Department of Mathematics and Computer Science, Kent, OH. 44240.
(5) 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 Transaction on Computers, Dec. 1987. Vol. C-36,no 12, pp 152-1538.
(6) Sharma N. and Wang Paul S., "Symbolic Derivation and Automatic Generation of Parallel Routines for Finite Element Analysis", appeared in Proceedings, ISSAC-1988, Roma, Italy, July 4-8, 1988.
(7) Weerawarana, Sanjiva and Wang, Paul S., "GENCRAY: User's Manual", Department of Mathematics and Computer Science, Kent State University, Kent, OH 44240.
(8) Balance Technical Summary, Sequent Computer Systems Inc.
(9) Bettess P. and Bettess J. A. "Automatic Generation of Shape Function Routines", Numerical Techniques for Engineering and Design, Proceedings of the International Conference on Numerical Methods in Engineering: Theory and Applications, NUMETA 1987, Swansea, paper S20, Vol II, 1987, Mantinus Nijhoff, Dordrecht.
(10) Aladjem, M. A., Mikhailov, M. D., "A Shape function Codewriter".
(11) Hughes, T. J. R., Ferencz, R.M. and Hallquyist, J.O., "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, pp 215-248, 1987.
(12) Levit Itzhak, "Element By Element Solvers of Order N", Computers and Structures, Vol 27, No. 3, pp. 357-360, 1987.
(13) Aho A. V., Sethi R., Ullman J. D., "Compilers Principles, Techniques, and Tool. 1988.
(14) Polychronopoulos Constantine D., "Parallel Programming and Compilers", Kluwer Academic Publishers, 1988.
(15) Coffman, E. G., Jr. (Ed.), "Computer and Job-Shop Scheduling Theory", John Wiley and Sons, NY 1976.
(16) Zienkiewicz, O. C., "The Finite Element Method in Engineering Science", Mc-Graw Hill, London, pp. 129-153.
(17) Carey F. G., Jiang B., "Element-By-Element Linear and Nonlinear Solution Schemes", Communications in Applied Numerical Methods, Vol. 2. 145-153. 1986.
(18) 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.
(19) Farhat, C., Crivelli, L., “A General Approach to Nonlinear FE Computations on Shared Memory Multiprocessors”, Internal Report CU-CSSC-87-09., Center for Space Structures and Control, College of Engineering, University of Colorado, Boulder, CO 80309.
(20) Winget, J. M., Hughes, T.J.R., “Solution Algorithms for Nonlinear Transient Heat Conduction Analysis Employing Element-By-Element Iterative Strategies.”, Computer Methods for Applied Mechanics and Enginnering, Vol. 52, 1985, pp 711-815.