



Pergamon

Computers Math. Applic. Vol. 35, No. 12, pp. 41–57, 1998

Published by Elsevier Science Ltd

Printed in Great Britain

0898-1221/98 \$19.00 + 0.00

PII: S0898-1221(98)00095-9

Using High Performance Fortran for Parallel Programming

G. SARMA AND T. ZACHARIA

Oak Ridge National Laboratory
Oak Ridge, TN 37831-6140, U.S.A.

D. MILES

The Portland Group, Inc.
Wilsonville, OR 97070, U.S.A.*(Received August 1997; October 1997)*

Abstract—A finite element code with a polycrystal plasticity model for simulating deformation processing of metals has been developed for parallel computers using High Performance Fortran (HPF). The conversion of the code from an original implementation on the Connection Machine systems using CM Fortran is described. The sections of the code requiring minimal inter-processor communication are easily parallelized, by changing only the syntax for specifying data layout. However, the solver routine based on the conjugate gradient method required additional modifications, which are discussed in detail. The performance of the code on a massively parallel distributed-memory Intel PARAGON supercomputer is evaluated through timing statistics. Published by Elsevier Science Ltd.

Keywords—Parallel program, High Performance Fortran, Finite element method, Deformation process simulation, Polycrystal plasticity.

1. INTRODUCTION

Metals have a polycrystalline microstructure, and over a broad range of processing conditions these polycrystals deform by shearing along crystallographic slip systems. Due to the limited modes of deformation available through slip, the crystals rotate under arbitrary deformations, with the reorientation proceeding in an ordered fashion. The anisotropic nature of single crystals is reflected in the macroscopic material properties primarily through the preferred arrangement of crystal orientations (known as crystallographic texture) [1,2]. Our research is focused at developing simulation capabilities to predict the evolution of texture during deformation processing of metals.

Simulating texture development and its impact on subsequent deformation requires material models which are capable of treating the distribution of crystal orientations and its evolution.

This research was sponsored by the Division of Materials Science, U.S. Department of Energy, under contract DE-AC05-96OR22464 with Lockheed Martin Energy Research Corporation. The research was supported in part by an appointment to the Oak Ridge National Laboratory Postdoctoral Research Associates Program administered jointly by the Oak Ridge National Laboratory and the Oak Ridge Institute for Science and Education. The authors acknowledge the use of the Intel PARAGON XP/S 35 located in the Oak Ridge National Laboratory Center for Computational Sciences (CCS), funded by the Department of Energy's Office of Scientific Computing, and the Connection Machine CM-5 at the National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign. Helpful discussions with R. Aggarwal and A. Beaudoin are gratefully acknowledged. The authors are grateful to M. Arnold and B. Radhakrishnan for reviewing the article.

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}\text{-T}\mathcal{E}\mathcal{X}$

Polycrystal models using the discrete aggregate representation of texture offer an effective means to address this challenge [3,4]. Each material point is typically associated with a collection of single crystals, characterized by their orientations and slip system hardness parameters. The response of the material point is derived from the averaged response of the aggregate. By placing an aggregate at each integration point of a finite element discretization of the workpiece, it is possible to model texture development during bulk deformation processes such as rolling, forging, extrusion, etc. [5–7].

Use of polycrystal plasticity models in conjunction with the finite element method requires significant computational resources. Assuming a viscoplastic constitutive law to model the shearing of slip planes leads to a nonlinear crystal constitutive relation, which must be solved for every crystal at each integration point to develop the material response. The nature of these calculations makes them ideal for implementation on parallel computer architectures, since they can be performed concurrently for all elements. This advantage has been exploited by using the Connection Machine systems, CM-200 and CM-5 (Thinking Machines Corp., Cambridge, MA), both for simulating industrial scale processes [8,9], as well as for detailed studies using idealized deformations [10,11].

While the code developed for the Connection Machine systems has been quite effective in treating large three-dimensional problems, it suffers from the drawback of being machine-specific. With the availability of other parallel computers from various vendors, it has become desirable to develop programs which can be easily ported across the different machines. The programming resources for writing efficient parallel programs have also seen a lot of improvement, with the development of message passing libraries [12] such as Parallel Virtual Machine (PVM) and Message Passing Interface (MPI), and with the introduction of High Performance Fortran (HPF) [13]. The latter, in particular, is well suited for developing portable versions of the code written using CM Fortran for the Connection Machines. Both CM Fortran and HPF are based on Fortran 90, with some differences in the syntax for specifying the distribution and layout of data. In addition, both languages use the data parallel programming model, with a single program controlling the distribution of data and the computations on them, and the compiler assuming responsibility for the actual distribution and movement of data among processors.

This article describes the work involved in developing a portable HPF version of a finite element code written for the Connection Machine systems using CM Fortran. In the next section, an outline of the mathematical and finite element formulation used by the code is given. Section 3 describes the implementation of the code on the Intel PARAGON using the *pghpf* compiler (The Portland Group, Inc., Wilsonville, OR). Efforts to optimize the solver using the conjugate gradient method are discussed, and the enhancements to improve I/O are noted. Some results based on the performance evaluation of various parts of the code are presented in Section 4, and illustrate the good parallel performance of the code. Closing remarks are presented in Section 5.

2. FINITE ELEMENT FORMULATION

2.1. Constitutive Model

The code for simulating the deformation of polycrystalline materials uses the finite element method along with a constitutive model based on polycrystal plasticity. The response at a material point is derived from the averaged response of an underlying aggregate of crystals. Elastic deformations are neglected, and plastic deformation is assumed to occur in each crystal by shearing along crystallographic slip systems. Due to the limited deformation modes available through slip, crystals must rotate to accommodate arbitrary deformations. The deformation gradient \mathbf{F}_c in the crystal can be written using a multiplicative decomposition as [4,5,14,15]

$$\mathbf{F}_c = \mathbf{R}^* \mathbf{F}_p, \quad (1)$$

where \mathbf{F}_p is the deformation gradient due to slip, and \mathbf{R}^* is the rotation of the crystal. Rewriting equation (1) in rate form leads to an expression for the crystal velocity gradient \mathbf{L}_c ,

$$\mathbf{L}_c = \dot{\mathbf{F}}_c \mathbf{F}_c^{-1} = \dot{\mathbf{R}}^* \mathbf{R}^{*\top} + \mathbf{R}^* \dot{\mathbf{F}}_p \mathbf{F}_p^{-1} \mathbf{R}^{*\top}, \quad (2)$$

or

$$\mathbf{L}_c = \dot{\mathbf{R}}^* \mathbf{R}^{*\top} + \mathbf{L}_p, \quad (3)$$

where \mathbf{L}_p represents the contribution to the crystal velocity gradient due to shearing along the slip planes, and may be expressed as a linear combination of the slip system shearing rates $\dot{\gamma}^{(\alpha)}$,

$$\mathbf{L}_p = \sum_{\alpha} \dot{\gamma}^{(\alpha)} \mathbf{T}^{(\alpha)} = \sum_{\alpha} \dot{\gamma}^{(\alpha)} \left(\mathbf{s}^{(\alpha)} \otimes \mathbf{n}^{(\alpha)} \right). \quad (4)$$

$\mathbf{T}^{(\alpha)}$ is the Schmid tensor for the α slip system, given by the dyadic product of the slip direction $\mathbf{s}^{(\alpha)}$ and the slip plane normal $\mathbf{n}^{(\alpha)}$. Denoting the symmetric and skew parts of $\mathbf{T}^{(\alpha)}$ by $\mathbf{P}^{(\alpha)}$ and $\mathbf{Q}^{(\alpha)}$, respectively, and separating the crystal velocity gradient into its symmetric and skew portions leads to expressions for the crystal rate of deformation \mathbf{D}_c ,

$$\mathbf{D}_c = \sum_{\alpha} \dot{\gamma}^{(\alpha)} \mathbf{P}^{(\alpha)}, \quad (5)$$

and the crystal spin \mathbf{W}_c ,

$$\mathbf{W}_c = \dot{\mathbf{R}}^* \mathbf{R}^{*\top} + \sum_{\alpha} \dot{\gamma}^{(\alpha)} \mathbf{Q}^{(\alpha)}. \quad (6)$$

Rewriting equation (6) results in the crystal reorientation rate $\dot{\mathbf{R}}^*$, given by the difference between the crystal spin and the plastic spin due to slip,

$$\dot{\mathbf{R}}^* = \left(\mathbf{W}_c - \sum_{\alpha} \dot{\gamma}^{(\alpha)} \mathbf{Q}^{(\alpha)} \right) \mathbf{R}^*. \quad (7)$$

If the crystal velocity gradient \mathbf{L}_c is known, it is only necessary to determine the partitioning of the shearing rates among the slip systems in order to update the orientation of the crystal lattice using equation (7).

For plastic deformations, assuming the volume to remain constant, a crystal must have at least five independent slip systems to accommodate arbitrary deformations (six independent components of rate of deformation tensor \mathbf{D}_c with the constraint that it must be traceless). High symmetry metals such as those with cubic crystal lattice typically have more than five available slip systems, leading to a possible nonuniqueness in the choice of active slip systems. This difficulty is overcome by assuming rate dependent slip, through a viscoplastic constitutive relation between the slip system shearing rate $\dot{\gamma}^{(\alpha)}$ and the resolved shear stress $\tau^{(\alpha)}$ [4,5,16,17],

$$\dot{\gamma}^{(\alpha)} = \dot{\gamma}_0 \left| \frac{\tau^{(\alpha)}}{\hat{\tau}} \right|^{1/m} \text{sign} \left(\tau^{(\alpha)} \right), \quad (8)$$

where m is the rate sensitivity parameter, and $\dot{\gamma}_0$ is a reference rate of shearing. $\hat{\tau}$ is a hardness parameter which represents resistance to plastic deformation. The resolved shear stress is the component of the traction along the slip direction, and is obtained using the Schmid tensor from the deviatoric Cauchy stress σ'_c ,

$$\tau^{(\alpha)} = \sigma'_c \mathbf{n}^{(\alpha)} \cdot \mathbf{s}^{(\alpha)} = \sigma'_c \cdot \mathbf{T}^{(\alpha)} = \sigma'_c \cdot \mathbf{P}^{(\alpha)}. \quad (9)$$

Eliminating $\dot{\gamma}^{(\alpha)}$ between equations (5) and (8), and using equation (9) for $\tau^{(\alpha)}$ leads to the following linearized expression for the crystal deformation rate in terms of the deviatoric stress

$$\mathbf{D}_c = \left[\sum_{\alpha} \frac{\dot{\gamma}_0}{\hat{\tau}} \left| \frac{\tau^{(\alpha)}}{\hat{\tau}} \right|^{(1/m)-1} \mathbf{P}^{(\alpha)} \otimes \mathbf{P}^{(\alpha)} \right] : \sigma'_c, \quad (10)$$

or

$$\mathbf{D}_c = \mathcal{S}_c : \sigma'_c, \quad (11)$$

where \mathcal{S}_c is a fourth-order crystal ‘‘compliance’’ tensor. The rate dependence of equation (8) permits inversion of equation (10), expressing the crystal deviatoric stress under a given deformation rate as

$$\sigma'_c = \mathcal{S}_c^{-1} : \mathbf{D}_c = \mathcal{S}_c^{-1} : \mathbf{D}. \quad (12)$$

Here the deformation rate in each crystal is assumed identical to the macroscopic value \mathbf{D} , under a Taylor hypothesis [3,4]. The crystal stresses are averaged to obtain the macroscopic value,

$$\sigma' = \frac{1}{N} \sum_c \sigma'_c = \frac{1}{N} \sum_c \mathcal{S}_c^{-1} : \mathbf{D} = \mathcal{C} : \mathbf{D}, \quad (13)$$

where \mathcal{C} is the average ‘‘stiffness’’ of the aggregate comprising N crystals.

2.2. Hybrid Formulation

Solution of the boundary value problem for material motion entails applying the balance laws for equilibrium and mass conservation, along with the constitutive assumptions discussed above. Following the approach described in Beaudoin *et al.* [10], a hybrid finite element formulation is employed for this purpose. The equilibrium statement is developed from a balance of tractions between all elements, and written as a weighted residual,

$$\sum_e \left[\int_{\partial \mathcal{B}_e} \Phi \cdot \mathbf{t}^{ie} dA - \int_{\partial \mathcal{B}_t} \Phi \cdot \mathbf{t} dA \right] = 0, \quad (14)$$

where $\partial \mathcal{B}_e$ is the element surface, $\partial \mathcal{B}_t$ is the portion of the element surface with applied tractions, \mathbf{t}^{ie} are inter-element tractions on contacting element faces, and \mathbf{t} are applied external tractions. Φ represents a weighting function which may be identified with the velocity field.

The Cauchy formula, $\mathbf{t}^{ie} = \sigma \mathbf{n}$, along with the divergence theorem in the first surface integral in equation (14) permits rewriting the weighted residual in terms of the Cauchy stress. Since the plastic deformation of the material is independent of the hydrostatic stress, it is convenient to separate the stress into its deviatoric (σ') and hydrostatic ($-p$) parts, leading to

$$\sum_e \left[\int_{\mathcal{B}_e} \nabla \Phi \cdot \sigma' dV - \int_{\mathcal{B}_e} p (\text{div } \Phi) dV + \int_{\mathcal{B}_e} (\text{div } \sigma) \cdot \Phi dV - \int_{\partial \mathcal{B}_t} \Phi \cdot \mathbf{t} dA \right] = 0, \quad (15)$$

where $\sigma = \sigma' - p\mathbf{1}$, p is the pressure and $\mathbf{1}$ is the second-order identity tensor.

The equation for mass conservation for the material assuming constant density reduces to a divergence-free velocity field, which is written using scalar weight ϑ as

$$\int_{\mathcal{B}_e} \vartheta (\text{div } \mathbf{u}) dV = 0. \quad (16)$$

The constitutive relation given by equation (13) is also written in weighted residual form using weighting function Ψ , after inverting the average stiffness,

$$\int_{\mathcal{B}_e} \Psi \cdot (\mathcal{C}^{-1} : \sigma' - \mathbf{D}) dV = 0. \quad (17)$$

Shape functions are introduced for the velocity \mathbf{u} , deviatoric stress σ' and pressure p , along with corresponding weighting functions Φ , Ψ , and ϑ , respectively, with the properties that the velocity shape functions provide continuous interpolation, the stress shape functions are piecewise discontinuous and linear in the global coordinates, and the pressure shape functions are constant. The shape functions for stress and pressure are chosen so as to satisfy $\text{div } \sigma = \mathbf{0}$ *a priori* at the element level, leading to the elimination of the third integral in equation (15).

The degrees of freedom corresponding to the crystal stresses are eliminated between equations (15) and (17), leading to a system of equations for the nodal velocities. Additional set of equations to solve for the velocity field are obtained from the incompressibility constraint (equation (16)). The velocity field is computed assuming the material state (crystal orientation \mathbf{R}^* and slip system hardness $\hat{\tau}$ values) and geometry to be fixed. Upon obtaining a converged solution, the state and geometry are updated. Equation (7) prescribes an evolution law for the orientation, while the hardness is updated using [5,18]

$$\dot{\hat{\tau}} = H_0 \left(\frac{\hat{\tau}_s - \hat{\tau}}{\hat{\tau}_s - \hat{\tau}_i} \right) \dot{\gamma}^*, \quad (18)$$

where hardening rate H_0 , initial hardness $\hat{\tau}_i$, and saturation hardness $\hat{\tau}_s$ are material parameters. $\dot{\gamma}^*$ is a measure of the net shearing rate on all the slip systems,

$$\dot{\gamma}^* = \sum_{\alpha} \left| \dot{\gamma}^{(\alpha)} \right|. \quad (19)$$

The key feature of the formulation, which enables effective implementation on a data parallel machine, lies in the use of piecewise discontinuous shape functions for the stresses. This allows for solution of the stresses at the element level using equation (17). The construction of the matrix

$$[H] = \int_{\mathcal{B}_e} [N^{\sigma}]^T [C^{-1}] [N^{\sigma}] dV,$$

where $[N^{\sigma}]$ are the stress interpolation functions, is performed concurrently for all elements. More importantly, its inversion is carried out in a sequence of data parallel operations for the in-place LU decomposition of a symmetric positive definite matrix, making this formulation efficient on a massively parallel architecture [10].

An important aspect of the implementation is the solution of the system of equations for the velocity field. Since direct solvers are difficult to optimize in a parallel computing environment, it is advantageous to use an iterative procedure, such as the conjugate gradient (CG) method [19]. In this context, enforcing the incompressibility constraint requires special attention, since it degrades the numerical condition of the resulting system of equations. This leads to a prohibitively large number of CG iterations in solving for the velocity field. The numerical performance is enhanced by improving the condition of the coefficient matrix through diagonal scaling, which is ideally suited for parallel implementation since it requires no communication among processors. In the formulation used by Beaudoin *et al.* [10], the incompressibility constraint is enforced by means of a modified consistent penalty approach, which seeks to decouple the solution for the pressure field from the conjugate gradient method. A detailed description of the formulation for enforcing incompressibility is discussed by Beaudoin *et al.* [8].

3. PARALLEL IMPLEMENTATION USING HPF

High Performance Fortran (HPF) provides a set of extensions to Fortran 90 to facilitate programming for parallel computers. HPF includes directives for distributing data across processors, for specifying parallel computations, and for invoking other programming paradigms from HPF [13]. These features permit parallel programming at a relatively high level, with the compiler assuming responsibility for the actual distribution and movement of data across processors.

The other attractive feature of HPF is the ability to write parallel code which is easy to port across different architectures.

The finite element formulation for modeling plastic deformation of metals was successfully implemented on the Connection Machine CM-200 and CM-5 using CM Fortran. The first task in converting the CM version to use HPF was to replace the `cmf$layout` specifications with `DISTRIBUTE` and `ALIGN` directives. In HPF, it is advantageous to use the `TEMPLATE` directive to declare an abstract array of indexed positions, and use the `DISTRIBUTE` directive to partition this template array. Using the `ALIGN` directive, other arrays can then be distributed with the same layout along particular axes. This approach is very convenient for parallelizing operations on arrays with the same layout. Use of CM Fortran in writing the original code, with Fortran 90 supported array syntax, meant only the specification of data layout needed modification. In the `cmf$layout` directive, no specification for a particular axis of an array defaults to `:serial`, which typically can be replaced by `BLOCK` in the `DISTRIBUTE` directive in HPF, although in some instances it is more advantageous to use a `CYCLIC` distribution. Similarly, `:serial` in a `cmf$layout` directive indicates no distribution of array elements along that particular axis, and is equivalent to `a *` in the `DISTRIBUTE` directive. Thus, layouts of the form:

```
cmf$layout a(:serial,), b()
```

would be changed to

```
!HPF$ DISTRIBUTE a(*,BLOCK)
!HPF$ DISTRIBUTE b(BLOCK)
```

Since the majority of arrays in the code were dimensioned either over the total number of elements or the total number of degrees of freedom, template arrays with these dimensions were introduced. These arrays were distributed across the processors, and other arrays were aligned with them.

```
!HPF$ TEMPLATE EL(0:maxel1)
!HPF$ DISTRIBUTE EL(BLOCK)
!HPF$ TEMPLATE DOF(0:maxdof1)
!HPF$ DISTRIBUTE DOF(CYCLIC)
```

Each processor thus receives a subset of the total number of elements (or degrees of freedom), and performs operations for them in a sequential fashion. Execution of the code with these preliminary modifications showed that most of the execution time was being spent in the CG solver for computing the velocity field. It must be noted here that the CM Fortran version of the code contained certain Connection Machine specific library calls, which were replaced with simple code. As will be discussed later, additional efforts were necessary to improve the efficiency of these parts of the code.

As discussed in the previous section, the choice of the interpolation functions for the velocity, stress, and pressure fields is designed to enable the stiffness matrix computations to proceed concurrently for all elements. This part of the formulation requires little or no communication between processors, and is therefore, easily implemented by distributing the elements among the available processors. Solution of the resulting system of equations poses a greater challenge, since each node gets contributions from several elements. If these elements are assigned to different processors, movement of data becomes unavoidable. The resulting communications calls are a source of pure overhead which work against parallelism.

3.1. Improving Efficiency of the CG Solver

As mentioned earlier, iterative solvers are more efficient on parallel machines, and the present formulation uses a preconditioned CG method [19]. The preconditioning operation is performed with the diagonal entries of the coefficient matrix. A vector is constructed from the inverse of these diagonal components, which is then distributed with the same layout as all other vectors

used in the CG solver. The preconditioning step then reduces to a product operation between corresponding entries of two vectors to yield a new vector, which requires no communication. The CG method consists of a series of vector additions and inner product computations, which are optimized by aligning the layout of the vectors involved. The most time consuming part is a sparse matrix vector multiply operation with the assembled global stiffness matrix. This part of the code required additional enhancements to get reasonable performance, and is discussed below.

The matrix vector product can be performed either on the assembled global stiffness matrix, or using the unassembled element stiffness matrices [20,21]. Here the second approach is adopted to avoid explicit assembly of the global matrix. The required product is computed using the following steps.

- *Gather*: The “gather” operation constructs element vectors from the global vector for each element. The correspondence between the global and element degrees of freedom is derived from the mesh connectivity array. It is convenient to place all entries of an element vector on the same processor. The gather operation requires data motion since the layout of the global degrees of freedom differs from that of the elements.
- *Local product*: After the gather, the full stiffness matrix and the vector for an element are available on the same processor. Each processor then computes this product for elements local to itself, and there is no communication needed.
- *Scatter*: The “scatter” operation forms the global product vector by adding contributions from the element vectors, using the mesh connectivity array to determine the proper locations. Since the global product vector has a different layout from the array containing element vectors, communication is necessary.

On the Connection Machine, the three steps listed above are optimized using routines from the Connection Machine Scientific Software Library (CMSSL). The gather and scatter operations are required by most finite element codes when moving between unassembled element and assembled global degrees of freedom. The pattern of data movement required for these operations remains unchanged if the mesh connectivity and the data distribution are the same. This fact is exploited by the CMSSL by providing a “setup” routine to compute the addresses on the processors and the routing information for data movement. The setup routine is called once to save this information, which is then used in any number of subsequent calls to gather (or scatter).

The CMSSL library calls were replaced by simple code to perform the three steps listed above. The poor condition of the system due to the incompressibility constraint, and the need for a large number of CG iterations for convergence has been discussed earlier. Improving the speed of the matrix vector product step was critical for obtaining a reasonable performance level from the code.

Due to the lack of a high-level math library in HPF, the local product step was optimized using a BLAS library call. Since these calls required use of code written in FORTRAN 77, they were implemented by declaring an interface with an extrinsic procedure. HPF provides the `EXTRINSIC` declaration to allow interface with programs written in other languages or styles [13]. A block of code declared to be `EXTRINSIC` is treated in the Single Program Multiple Data (SPMD) programming style, with each processor executing the same code with data local to that processor. Communication using a message passing library is handled explicitly by the programmer. Each processor called the BLAS routine `dgemv` to compute the product for the set of elements local to that processor. Shown in Figure 1 are sections of the subroutine `sparse_matvec_ebe`, illustrating declaration of the extrinsic procedure `loc_matrix_vector_mult`, and the call to this procedure.

The code for the extrinsic procedure is shown in Figure 2. Since the code had to be written in FORTRAN 77, the dimensions of the arrays had to be passed as arguments to the local routine. For an arbitrary problem, the number of elements assigned to each processor needs to be determined at run time. The NX library call `mynode()` [22] paragon has been used to deal with

```

subroutine sparse_matvec_ebe(y1, x1, stiffness, bcs, nodes, dofloc)
c   Compute the product [y1] = [stiffness]*[x1]
   use hpf_library
   ...
   real*8 y1(0:maxdof1), x1(0:maxdof1)
!HPF$ ALIGN (:) WITH DOF(:) :: x1, y1
   real*8 stiffness(0:kdim1, 0:kdim1, 0:maxel1)
!HPF$ ALIGN (*,*,:) WITH EL(:) :: stiffness
   real*8 t1(0:kdim1, 0:maxel1), t2(0:kdim1, 0:maxel1)
!HPF$ ALIGN (*,:) WITH EL(:) :: t1, t2
   ...
   interface
   ...
   extrinsic (f77_local)
1   subroutine loc_matrix_vector_mult(y, a, x, kdim, nepp, procem,
2   nerem)
   real *8 y(:,,:), a(:,,:), x(:,,:)
!HPF$   ALIGN (*,*,:) WITH EL(:) :: a
!HPF$   ALIGN (*,:) WITH EL(:) :: y, x
   integer kdim, nepp, procem, nerem
   end subroutine loc_matrix_vector_mult
   end interface
c   First perform gather operation from x1 to t1
   ...
c   Compute the element product vectors
   t2 = 0.0
   call loc_matrix_vector_mult(t2, stiffness, t1, kdim, nepp,
1   procem, nerem)
c   Perform scatter operation to form global product vector y1 from t2
   y1 = 0.0
   y1 = sum_scatter(t2, y1, nodes)
   where( bcs ) y1 = 0.0
   return
end

```

Figure 1. HPF code for matrix vector multiply.

arbitrary sized problems for the Intel PARAGON implementation. This is a machine specific call which returns the abstract processor number for that processor. Alternately, it is possible to write a wrapper routine which determines the size of the subarray on each processor, which can then be used in calling the extrinsic procedure.

The HPF library procedure call `sum_scatter` was used to optimize the scatter operation. Efficiency of the gather operation was improved using extrinsic subroutines. The gather operation assigns values from the global vector to element vectors using the nodal connectivity. Since the global vector is distributed, an extrinsic routine is called first to collect the full vector and make it available to each processor. Another extrinsic routine then assigns values from the full vector to the element vectors local to that processor. Assembling the full vector on each processor was accomplished by a single call to the `gcolx` procedure available as part of the NX message passing library on the Intel PARAGON [22]. It is convenient to use the CYCLIC distribution for the vector, although BLOCK distributed vectors can also be assembled with minor changes. The assignment of values is then performed in parallel by each processor for a subset of elements which are local

```

subroutine loc_matrix_vector_mult(y, a, x, kdim, nepp, procem,
1  nerem)
  implicit none
  integer kdim, nepp, procem, nerem
  real *8 y(kdim, nepp), x(kdim, nepp), a(kdim, kdim, nepp)
  include 'fnx.h'
  integer i, n
  n = nepp
  if(mynode().gt.procem) n = nerem
  do i=1,n
    call dgemv('n', kdim, kdim, 1.d0, a(1,1,i), kdim, x(1,i),
1      1, 0.d0, y(1,i), 1)
  end do
  return
end

```

Figure 2. Local code for element level matrix vector product computations.

to it. Once again, the NX call `mynode()` is used to compute the global element number for a given local element number on each processor. The local routines for the gather operation are shown in Figure 3.

```

subroutine loc_collect(y, x, maxlen, ndp)
  implicit none
  include 'fnx.h'
  include 'local.inc'
  integer maxlen, ndp
  real *8 y(maxlen), x(ndp)
  call gcolx(x(1), xlen, y(1))
  return
end

subroutine loc_gather(y, x, p, maxdof, maxel, nepp, procem,
1  nerem, kdim)
  implicit none
  include 'fnx.h'
  integer maxdof, maxel, kdim, nepp, procem, nerem
  real *8 y(maxdof), x(kdim, nepp)
  integer p(kdim, maxel)
  integer i, j, npi, mystart, np
  np = nepp
  if(mynode().gt.procem) np = nerem
  mystart = mynode()*nepp
  do i=1, np
    npi = mystart + i
    do j=1, kdim
      x(j, i) = y(p(j, npi))
    end do
  end do
  return
end

```

Figure 3. Local routines for the gather operation.

3.2. Improving Efficiency of I/O

In addition to the changes described above, it was necessary to implement an efficient I/O scheme. HPF does not address parallel I/O, and since the original code was written for the Connection Machine, the `read` and `write` statements contained explicit indexing of the arrays using `DO` loops. Initial attempts to run large problems indicated an almost linear increase in the time spent for I/O with number of processors. The code was altered by placing the segments dealing with I/O in extrinsic procedures, so that each processor executed them simultaneously. Additional speedup was obtained by placing the data files in the Parallel File System (PFS) available on the Intel PARAGON [22].

4. PERFORMANCE EVALUATION

The modified code with HPF directives was implemented to run on a 512-processor Intel PARAGON. Various trials were conducted to examine the effect of introducing HPF directives, as well as the changes made to improve the solver and the I/O. The results of these trial runs are presented, along with some comparisons to highlight the differences from the CM-5 version.

The introduction of local routines to speed up the matrix vector product in the CG method are considered first. A finite element discretization consisting of a $4 \times 4 \times 4$ mesh of eight-noded brick elements was deformed in plane strain compression. These simulations were carried out with the improved I/O calls using local routines, so as to speed up the process of obtaining the timing data presented in Table 1. Since the element stiffness computations require minimal communication among processors, they scale very well with number of processors. Doubling the number of processors cuts the time for the stiffness computations in half. When the number of processors is very small, the reduction in time is more than 50%. This is probably due to some overhead associated with the HPF directives, leading to increased time on a single processor even when there is no distribution of data. The gather and scatter operations take up more time with increasing number of processors. As shown in Table 1, the time spent in the CG solver increases with processors, and this serves to offset the decrease in time for stiffness computations. The result is an increase in total execution time with processors, indicating a need to improve the performance of the solver.

Table 1. Performance for 64 element problem prior to optimizing the matrix vector product in the CG method.

Number of processors	1	2	4	8
Time for stiffness computations [s]	245.2	102.1	45.3	25.7
Time for CG solver [s]	53.6	289.9	543.9	1010.4
Total execution time [s]	309.8	400.6	596.8	1047.4

The performance of the same problem was examined after modification of the scatter, gather, and local product operations, in that order. The results of the various modifications are summarized in Table 2. The numbers shown in Table 1 indicate that the total execution time is essentially the sum of the time spent in the stiffness computations and in the CG solver, with other parts of the code contributing very little time. In what follows, we concentrate on the time spent in the CG solver to study the effects of the changes.

Table 2. Performance of 64 element problem after optimizing the matrix vector product in the CG method. The numbers indicate time spent in the CG solver in seconds.

Number of processors	1	2	4	8	16	32
After introducing <code>sum_scatter</code>	115.4	62.2	37.9	29.4	38.9	89.6
After introducing local gather	115.3	61.5	36.2	24.9	22.7	28.5
After introducing BLAS	85.3	52.2	32.1	23.4	21.6	28.1

Introduction of the HPF library call `sum_scatter` leads to a significant improvement in the CG method (compare times in Table 2 with those for the CG solver in Table 1). On a single processor, the time spent in the CG method actually goes up, again probably due to some overhead associated with the HPF calls. With increasing processors, the time for the solver goes down up to eight processors. When the problem is run on 16 or 32 processors, the gather routine starts to dominate, and the gain in using `sum_scatter` is offset by increase in time for gather.

Use of local routines to perform the gather operation shows no significant effect for small number of processors. However, for 16 or 32 processors, there is a significant reduction in the time, indicating a need for these routines when using large number of processors. The influence of using BLAS routines is seen in the reduction of time when only one or two processors are used. The local product is performed without need for communication, and the BLAS routines merely add speed to the calculations. When the 64 element problem is run on 16 (or 32) processors, there are only four (or two) elements on each processor. The time spent for the product computations is so small that gain in using BLAS routines is not clearly seen. However, the BLAS call is about six times faster than a simple DO loop, and this can become a significant factor as the problem size increases.

The next set of trial simulations were conducted using a $10 \times 10 \times 10$ mesh of 1000 hexahedral elements (1331 nodes) to examine the scale up for a larger problem. Figure 4 shows the total execution time as a function of number of processors. The execution time is plotted using a logarithmic scale to better indicate the reduction in time at larger numbers of processors. While this is not a particularly large problem, the decrease in time up to 50 processors (for which case the number of processors well exceeds the number of elements per processor) is noteworthy. As the number of processors is increased further, the overhead for communication calls increases relative to the computations, leading to an increase in the total time.

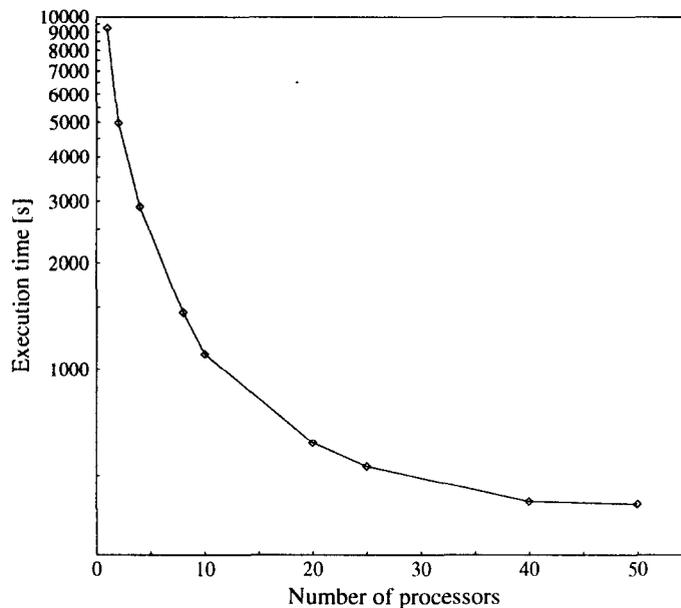


Figure 4. Execution time for 1000 element problem with number of processors.

The aspect of I/O was also examined using the same problem, by comparing time required for execution with no special changes to handle I/O against time required when using local routines for I/O. The resulting plot is depicted in Figure 5, and shows that the total time is larger when local routines are not called. In addition, it is observed that the total time starts

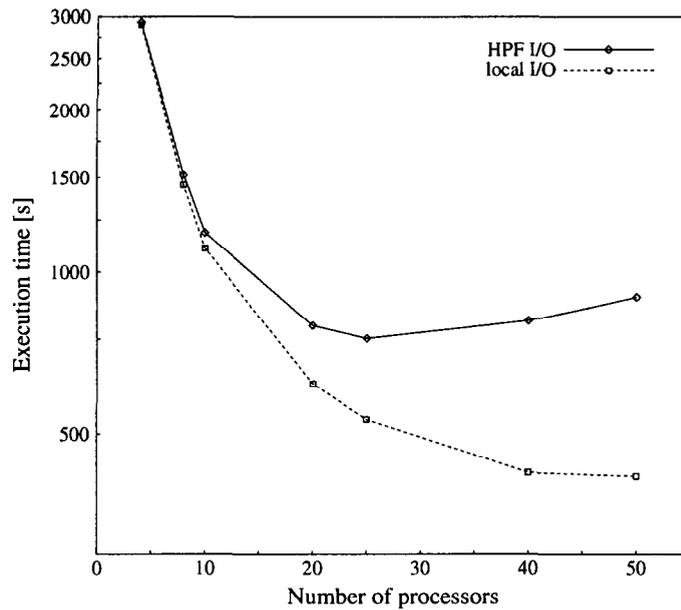


Figure 5. Comparison of execution time for 1000 element problem with and without local routines for I/O.

to increase beyond 25 processors, unlike the case when local calls are used. Examination of the timing data indicates a more or less linear increase in the time spent for I/O using regular HPF code. Additional speedup of the I/O operations was obtained by placing the data files in the Parallel File System (PFS) available on the Intel PARAGON. The gains in using PFS became more obvious when using greater number of processors to handle bigger problems. It must be mentioned here that the I/O statements used explicit indexing of arrays, which is quite efficient on the CM-5. Use of I/O calls with entire arrays would lead to better performance in HPF.

Figure 6 shows the variation of total execution time with number of processors for a 4096 element mesh deformed in plane strain compression over a single strain increment. Once again there is good scale up of the problem up to 128 processors (32 elements per processor). For comparison, the execution times for the same problem using the CM Fortran version of the code on the Connection Machine CM-5 are shown in Figure 7. It must be noted here that the HPF version of the code computes the velocity field using an isotropic constitutive model to generate an initial guess for the subsequent computation using the texture model discussed in Section 2.1. The CM-5 version does not contain the velocity calculation using the isotropic model. The initial calculation took approximately 30% of the total computation time on the PARAGON. Taking this into account, it is seen that the HPF code takes five to seven times longer than the CM Fortran version on the same number of processors. The difference in the processor speeds on the two machines could be partly responsible for the observed trend. However, timing of the matrix vector multiply routines on the two machines shows that for roughly the same number of calls, the HPF code takes about ten times longer than the CM Fortran version. Since the local BLAS routines permit relatively quick calculation of the element product vectors, the difference in the times can be traced to the gather and scatter operations, clearly showing the influence of the CMSSL library calls in achieving better performance on the CM-5.

Figures 8 and 9 show the relative amount of time spent in the CG solver and the matrix vector multiply routines for the 1000 and 4096 element meshes, respectively. Curve 'a' in these figures indicates the variation of the time spent in the matrix vector multiply routine as a percent of the total time required by the solver on different processors. Initially, a large part of the solver time is

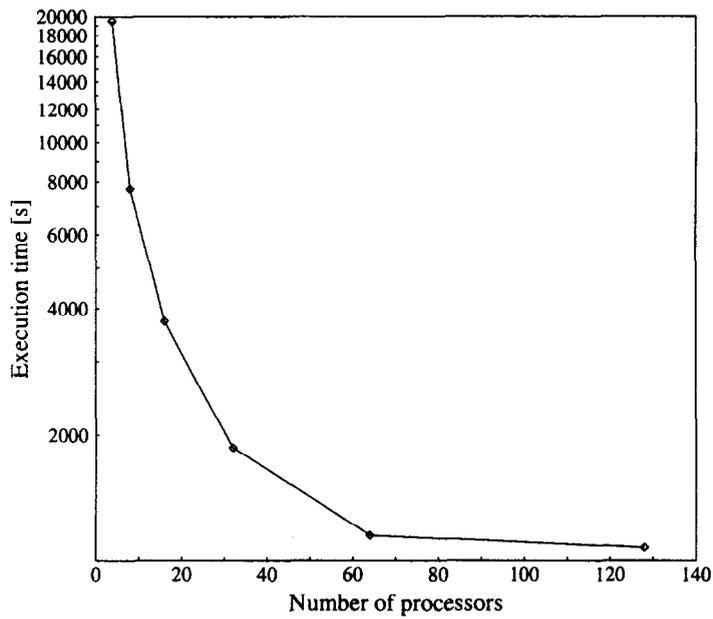


Figure 6. Execution time for 4096 element problem with number of processors on the Intel PARAGON.

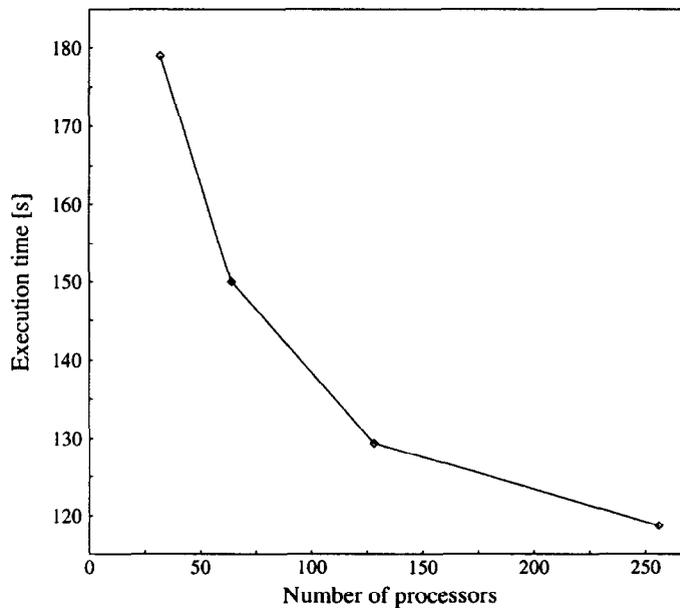


Figure 7. Execution time for 4096 element problem with number of processors on the Connection Machine CM-5.

devoted to the product calculation. With increasing number of processors, the time required by the inner product computations also goes up, since there is greater communication requirement. Optimizing the computation of a global sum operation then becomes important. Also shown in Figures 8 and 9 is the time spent in the solver routine as a fraction of the total time for all the computations (excluding I/O). This curve (labeled 'b') shows that for most part, the solver takes up around 85% of the computational time. This fraction is closer to 50% on the CM-5.

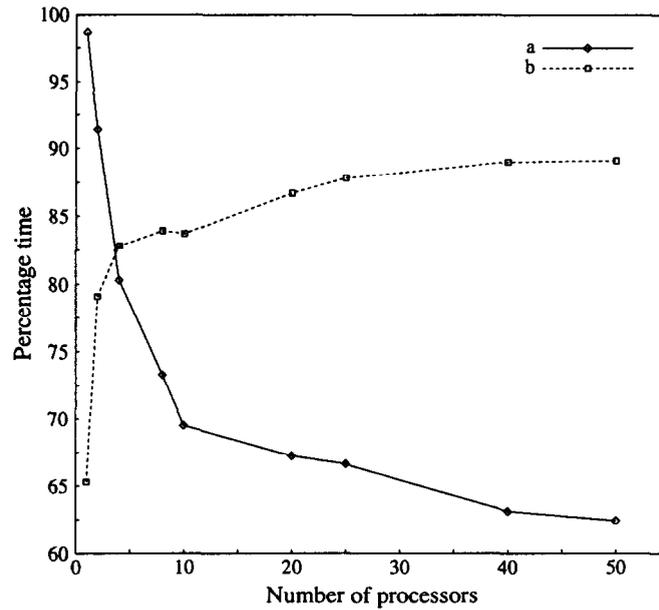


Figure 8. Fractional times for 1000 element problem with number of processors on the Intel PARAGON. Curve 'a' represents time spent in the matrix vector multiply routine as a fraction of time for solver. Curve 'b' is the time spent in the solver as a fraction of the total time for computations.

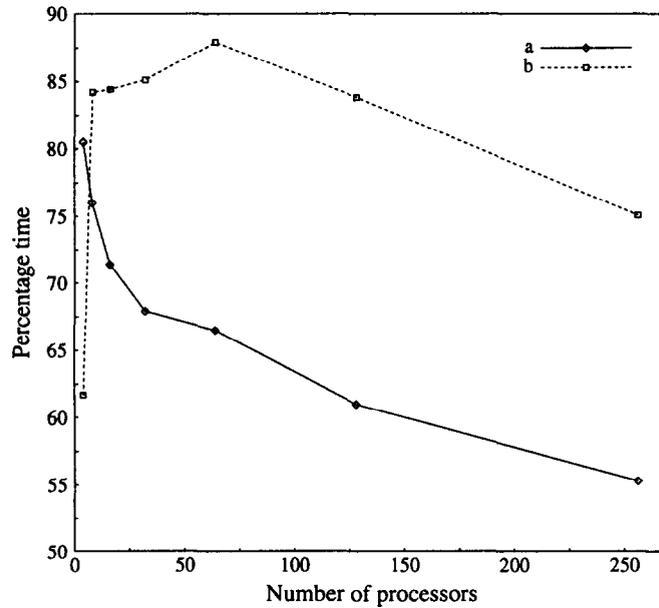


Figure 9. Fractional times for 4096 element problem with number of processors on the Intel PARAGON. Curve 'a' represents time spent in the matrix vector multiply routine as a fraction of time for solver. Curve 'b' is the time spent in the solver as a fraction of the total time for computations.

For the examples considered here, the stiffness calculations required very little time since each element contained only one crystal. With more crystals per element, there will be a shift in the computational load from the solver to the stiffness calculations.

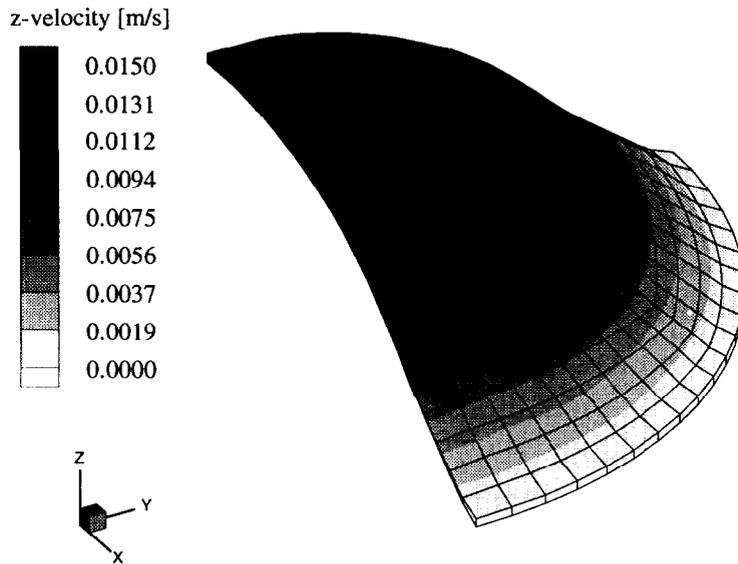


Figure 10. Variation of velocity component along the z -axis for the sheet stretching problem.

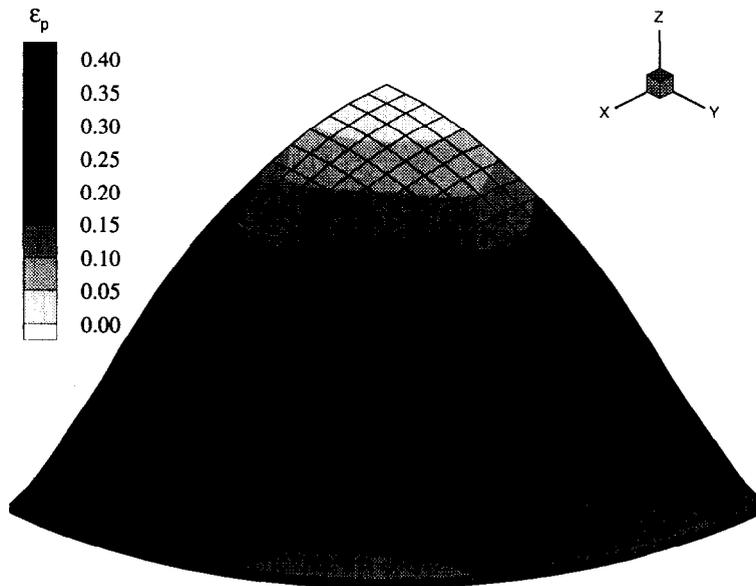


Figure 11. Variation of the effective plastic strain for the sheet stretching problem.

As an application of the code for large scale deformation processing, a sheet stretching problem was simulated using a single layer of 300 eight-node brick elements. Each element contained 256 crystals, which were initialized with orientations corresponding to a rolling texture generated from experimental measurements [23]. The deformed mesh with the z -velocity contours is shown in Figure 10, and the effective plastic strain contours are depicted in Figure 11. The effective plastic strain is computed as the norm of the accumulated strain during the deformation. The nonuniform straining due to texture-driven anisotropy is clearly seen. The sheet has been observed to fail along the x -axis, which is the rolling direction in the original sheet, during limiting dome height tests. The maximum value of the effective plastic strain occurs along the x -axis, indicating the possibility of failure at this location, which is consistent with the experimental observations [24].

The simulation results shown here were generated using the HPF code running on a 30 node partition of the Intel PARAGON. A total of 80 strain increments took 25 hours for this run. The time spent in the stiffness computations was twice as much as the time spent in the solver. A similar simulation with a different initial texture was carried out in two stages. The first 30 increments were conducted on a 100 node partition and took about 8.8 hours, with the solver taking almost four times as long as the stiffness computations. The remaining 50 increments were simulated on a 60 node partition, with the solver taking roughly the same amount of time as the stiffness calculations, with a total time of 7.3 hours. The same problem on the CM-5 took 1.7 hours on a 32 node partition, and on a 4 node partition of an SGI Power Challenge with an MPI implementation using the petSC solver, it completed in 24.7 hours [25].

5. CONCLUDING REMARKS

A finite element formulation based on a polycrystal plasticity constitutive model has been developed for parallel computers using High Performance Fortran (HPF). The development was based on converting a CM Fortran version to utilize HPF directives for data distribution and layout. The Fortran 90 style of programming used in the original code facilitated the conversion process, requiring minimal changes for most part. HPF permits parallel programming at a relatively high level, by providing directives to specify the layout and distribution of arrays at a global level. The actual distribution and movement of data, and synchronization of processors is handled by the compiler.

While layout directives permit easy parallelization of some calculations, the efficiency of the code can be poor for operations which require data movement across processors. Stiffness computations were easy to parallelize, by spreading the elements across processors. However, the solver for the system of equations required greater effort. Most finite element codes utilize “gather” and “scatter” operations to relate quantities between the unassembled element level and the assembled global level. For the implementation discussed here, the solution of the equations was obtained using the conjugate gradient (CG) method, with an element-by-element approach for the matrix vector multiply. Gather and scatter operations were required for each CG iteration, and due to the incompressibility constraint, the number of such iterations was quite large. The solver took up close to 85% of the computational time, with the gather/scatter calls accounting for the bulk of it. Comparisons with trials on the CM-5 indicate that the lack of a provision to reuse the traces for data movement might be contributing to the large time requirement for these operations. The current HPF language standard does not require reuse of schedules by the compiler. It is hoped that future revisions of the standard will address this issue.

The other main issue which requires attention in HPF is parallel I/O. Use of local calls with the `EXTRINSIC` feature enabled faster I/O on the Intel paragon. It was also possible to place the data files in the Parallel File System on the PARAGON to obtain better performance, but similar devices may not be available on other platforms. In addition to the I/O, the `EXTRINSIC` directive also proved useful in improving the efficiency of certain parts of the solver routine. The `pghpf` compiler used for the work described in this article provides support only for extrinsic code written in FORTRAN 77. Consequently, some extra coding was required to determine the extents of subarrays on each processor when using the extrinsic routines. In this context, availability of support for `HPF_LOCAL` would be quite useful.

A major advantage of using HPF is the ability to write portable programs. The code developed for the Intel PARAGON was optimized with a few machine-specific calls based on the NX message passing library to speed up the extrinsic routines. Minor modifications to remove these calls were sufficient to permit compilation and execution of the same code on the SGI Power Challenge Onyx and the IBM SP/2.

The code conversion used in this study was achieved with the minimum of changes from the CM Fortran version. However, as applications are ported from the CM-5 to other machines, it

may become necessary to alter some of the routines which depend heavily on the CMSSL library calls for efficient parallel performance. For instance, in relation to the solver routine discussed here, it may be more appropriate to work with an assembled system instead of the element-by-element approach for the matrix vector multiply. Such considerations apart, HPF provides a useful alternative to the message passing route to writing parallel programs. It is hoped that future developments of the compiler will address some of the issues such as the need for a better communications library and parallel I/O discussed here.

REFERENCES

1. U.F. Kocks, Constitutive relations for slip, In *Constitutive Equations for Plasticity*, (Edited by A.S. Argon), pp. 81–115, MIT Press, Cambridge, MA, (1975).
2. U.F. Kocks, Constitutive behavior based on crystal plasticity, In *Unified Constitutive Equations for Creep and Plasticity*, (Edited by A.K. Miller), pp. 1–88, Elsevier Applied Science, New York, (1987).
3. G.I. Taylor, Plastic strain in metals, *J. Inst. Metals* **62**, 307–324, (1938).
4. R.J. Asaro and A. Needleman, Texture development and strain hardening in rate dependent polycrystals, *Acta Metall.* **33** (6), 923–953, (1985).
5. K.K. Mathur and P.R. Dawson, On modeling the development of crystallographic texture in bulk forming processes, *Int. J. Plast.* **5**, 67–94, (1989).
6. K.K. Mathur and P.R. Dawson, Texture development during wire drawing, *J. Engng. Mater. Technol.* **112**, 292–297, (1990).
7. S.R. Kalidindi, C.A. Bronkhorst and L. Anand, Crystallographic texture evolution in bulk deformation processing of FCC metals, *J. Mech. Phys. Solids* **40** (3), 537–569, (1992).
8. A.J. Beaudoin, P.R. Dawson, K.K. Mathur, U.F. Kocks and D.A. Korzekwa, Application of polycrystal plasticity to sheet forming, *Comput. Methods Appl. Mech. Engrg.* **117**, 49–70, (1994).
9. A. Kumar and P.R. Dawson, Polycrystal plasticity modeling of bulk forming with finite elements over orientation space, *Comp. Mech.* **17** (1–2), 10–25, (1995).
10. A.J. Beaudoin, P.R. Dawson, K.K. Mathur and U.F. Kocks, A hybrid finite element formulation for polycrystal plasticity with consideration of macrostructural and microstructural linking, *Int. J. Plast.* **11** (5), 501–521, (1995).
11. G.B. Sarma and P.R. Dawson, Effects of interactions among crystals on the inhomogeneous deformations of polycrystals, *Acta Mater.* **44** (5), 1937–1953, (1996).
12. O.A. McBryan, An overview of message passing environments, *Parallel Computing* **20**, 417–444, (1994).
13. C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele, Jr. and M.E. Zosel, *The High Performance Fortran Handbook*, MIT Press, Cambridge, MA, (1994).
14. J.R. Rice, Inelastic constitutive relations for solids: An internal-variable theory and its application to metal plasticity, *J. Mech. Phys. Solids* **19**, 433–455, (1971).
15. D. Pierce, R.J. Asaro and A. Needleman, Material rate dependence and localized deformation in crystalline solids, *Acta Metall.* **31**, 1951–1976, (1983).
16. J.W. Hutchinson, Bounds and self-consistent estimates for creep of polycrystalline materials, *Proc. R. Soc. Lond. A* **348**, 101–127, (1976).
17. J. Pan and J.R. Rice, Rate sensitivity of plastic flow and implications for yield surface vertices, *Int. J. Solids Struct.* **19**, 973–987, (1983).
18. U.F. Kocks, Laws for work-hardening and low-temperature creep, *J. Eng. Mater. Tech.* **98**, 76–85, (1976).
19. R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Press, (1994).
20. K.K. Mathur, Parallel algorithms for large scale simulations in materials processing, In *Simulations of Materials Processing: Theory, Methods and Applications (NUMIFORM 95)*, (Edited by S.-F. Shen and P.R. Dawson), pp. 109–114, Balkema, Rotterdam, (1995).
21. Z. Johan, T.J.R. Hughes, K.K. Mathur and S.L. Johnsson, A data parallel finite element method for computational fluid dynamics on the Connection Machine system, *Comp. Meth. Appl. Mech. Engrg.* **99**, 113–134, (1992).
22. *Paragon User's Guide*, Intel Corporation, (1993).
23. A.J. Beaudoin, private communication, (1996).
24. B. Ren, J.G. Morris and A.J. Beaudoin, Influence of crystallographic texture on the biaxial stretchability of AA 5182 sheet material, *J. of Metals* **48** (6), 22–25, (1996).
25. A.J. Beaudoin, J.D. Bryant, P.R. Dawson and D.P. Mika, Incorporating crystallographic texture in finite element simulations of sheet forming, *Proc. NUMISHEET '96* (to appear).