Developing a Geodynamics Simulator with PETSc

Matthew G. Knepley[1], Richard F. Katz[2], and Barry Smith[1]

[1] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, [knepley,bsmith]@mcs.anl.gov
[2] Department of Earth and Environmental Sciences, Lamont Doherty Earth Observatory, Palisades, NY, katz@ldeo.columbia.edu

Summary. Most high-performance simulation codes are not written from scratch but begin as desktop experiments and are subsequently migrated to a scalable, parallel paradigm. This transition can be painful, however, because the restructuring required in conversion forces most authors to abandon their serial code and begin an entirely new parallel code. Starting a parallel code from scratch has many disadvantages, such as the loss of the original test suite and the introduction of new bugs. We present a disciplined, incremental approach to parallelization of existing scientific code using the PETSc framework. In addition to the parallelization, it allows the addition of more physics (in this case strong nonlinearities) without the user having to program anything beyond the new pieces of discretization code. Our approach permits users to easily develop and experiment on the desktop with the same code that scales efficiently to large clusters with excellent parallel performance. As a motivating example, we present work integrating PETSc into an existing plate tectonic subduction code.

1 Geodynamics of Subduction Zones

Subduction zones, where one of the Earth's surface plates collides with another and sinks into the deep mantle (Fig. 1a) [20], are the locus of many of the world's most devastating natural disasters, especially volcanic eruptions and earthquakes. Subduction zone volcanism such as the 1980 eruption of Mount St. Helens is characterized by violent explosions of ash and rock. Despite the relevance of this type of volcanism to problems ranging from public safety to global climate change and mass-extinction events in the geologic record, a detailed understanding of its source is lacking. Clues are abundant, however, in the rocks erupted from subduction zone volcanos which record a history of formation, transport, and eruption in their distinct geochemistry. The complexity of these processes in terms of the governing reactive thermochemical fluid dynamics and non-Newtonian rheology is significant; simplified models have proven inadequate in explaining basic observations [18].

More sophisticated PDE-based computational models are needed to address the sharp nonlinearities typically unexplored in past work. The large separation in length and time scales of the constituent physics implies a great computational cost due to the need for fine meshes to resolve the small (but relevant) scales. Moreover, the

highly nonlinear character of the non-Newtonian, temperature-dependent viscosity demands costly solution algorithms. This high computational complexity coupled with long development times have put such models out of reach for most geoscientists. The Portable Extensible Toolkit for Scientific Computation (PETSc) [5] makes it easier for geoscientists to overcome these barriers. In an abstract sense, PETSc provides a framework for collaboration between geoscientists with complex modeling problems and numerical analysts and software engineers who have the encapsulated numerical methods for solving those problems. In the case described here, the original model was a specialized, single linear solver serial code capable of calculating the thermal structure due to an analytically prescribed isoviscous flow field. By porting this code into the PETSc framework we were able to extend it to solve for fully coupled thermal structure and non-Newtonian flow with a choice of many scalable parallel solvers, flexible boundary conditions, convenient parameter input, real-time code steering, and many other features not available with the serial version. This was done using an incremental process by first replacing the custom linear solver with PETSc's general purpose nonlinear solver, then replacing the sequential data structures with PETSc's parallel ones and then finally adding the additional nonlinear physics (to the portion of the code that discretizes the PDE).

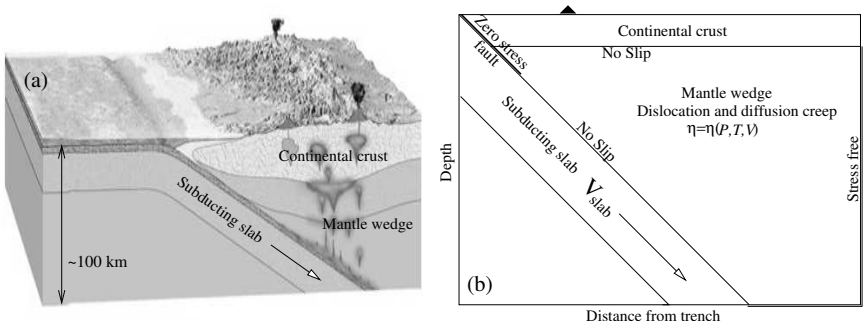

Fig. 1. (a) Schematic diagram of a subduction zone. Oceanic lithosphere is colliding with continental lithosphere and being subducted. Volatile compounds are released from the subducting slab at depth and enter the mantle wedge, lowering the melting temperature of the rock and causing partial melting. This melt rises to feed volcanos at the surface. **(b)** Schematic diagram of the computational domain. Boundary conditions on the flow field are imposed at the bottom of the crust, the top of the slab, and the intersection of the mantle wedge and the domain boundary. Flow velocity within the mantle wedge is the solution to equation (1). Potential temperature throughout the domain is the solution to equation (3). The "zero stress fault" represents a frictional sliding surface. On geologic time scales all stress on this boundary is relieved in earthquakes and does not cause deformation.

Any model of subduction zone volcanism must be based on the thermal and flow structure of the slowly moving solid mantle wedge, shown in Fig. 1. While the magnesium and iron-rich mantle rock is solid on human time-scales (as evidenced by the seismic waves it transmits), on geologic time-scales it undergoes two modes of

solid-state creep [14]. Its motion can be described by the Stokes equation for steady flow of an incompressible, highly viscous fluid (with zero Reynolds number),

$$\boldsymbol{\nabla} P = \boldsymbol{\nabla} \cdot \left[\eta \left(\boldsymbol{\nabla} \mathbf{V} + \boldsymbol{\nabla} \mathbf{V}^T \right) \right]; \quad s.t. \ \boldsymbol{\nabla} \cdot \mathbf{V} = 0, \tag{1}$$

$$\eta = \left(1/\eta_{disl} + 1/\eta_{difn} \right)^{-1}, \tag{2}$$

where P is the fluid pressure, $\mathbf{V}$ is the mantle velocity field, η_{difn} is the diffusion creep viscosity and η_{disl} is the dislocation creep viscosity. Both creep mechanisms give an Arrhenius-type dependence on pressure and temperature. Dislocation creep has an additional non-Newtonian strain rate dependence. The strength of the nonlinearity of viscosity makes this equation difficult to solve without a good initial guess. In our code the initial guess in provided by a continuation method: we modify equation (1) to let $\eta \rightarrow \eta^{\alpha}$, where α is a number between zero and one. The continuation method, described in section 5.1, varies α over a sequence of solves of increasing nonlinearity to reach natural variation in viscosity. The solution at each step is used as a guess for the following solve. The first step in this process is to solve the problem for constant viscosity (α=0), where the system of equations is linear and analytically tractable.

The production of molten rock depends fundamentally on the mantle temperature field. The distribution of heat is governed by the conservation of enthalpy, expressed as an advection-diffusion equation,

$$\frac{\partial \theta}{\partial t} + \mathbf{V} \cdot \boldsymbol{\nabla} \theta = \kappa \nabla^2 \theta, \tag{3}$$

where θ is the mantle potential temperature and κ is the thermal diffusivity of the mantle. The mantle has a very low thermal diffusivity compared to materials such as metal or water, and thus the advection term dominates in equation (3).

For $0 < \alpha \leq 1$, equations (1) and (3) form a nonlinear set coupled through the viscosity. The solution of these equations is the first stage in modeling magma genesis in subduction zones. Future work will incorporate equations of porous flow of volatiles and magma through the mantle, reactive melting, and geochemical transport. Even without these complexities, however, we have achieved interesting results with the simple, though highly nonlinear, set of equations given above. Some of these results are presented below, after an in-depth look at the application development process in the PETSc framework.

2 Integrating PETSc

PETSc is a set of library interfaces built in a generally hierarchical fashion. A user may decide to use some libraries and disregard others. Fig. 2 illustrates this hierarchy of dependencies. For instance, a user can use only the PETSc linear algebra (vectors and matrices) libraries, or add linear solvers to those, or add nonlinear solvers to the entire group. Thus, integration may proceed in several stages, which we discuss in the following sections. The first step is to incorporate the PETSc libraries into the

existing compile system (i.e., `make`) and to initialize the PETSc runtime. This may be done in two ways.

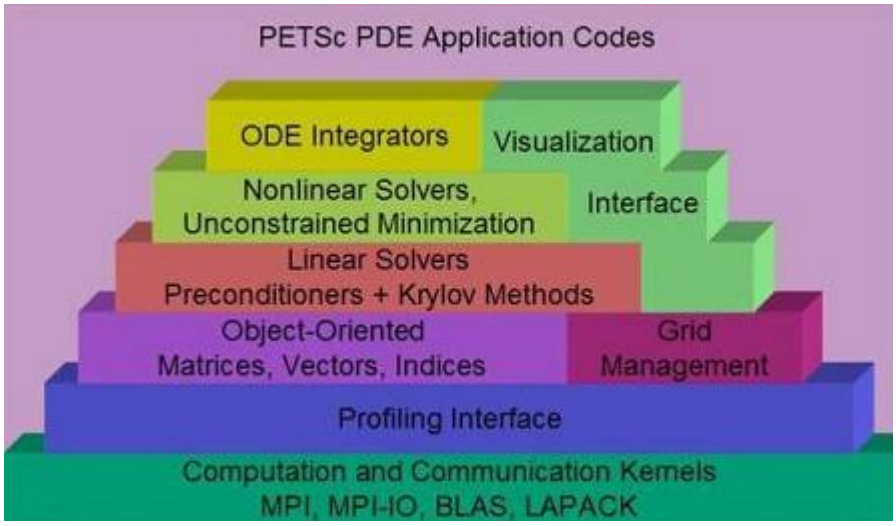

Fig. 2. Interface hierarchy in a PETSc application.

The simplest approach is to adopt the PETSc makefile structure, which is presented in Fig. 3 for an application using the nonlinear solver package. The user includes the `bmake/common/base`, which defines both rules for compiling source and variables that are used during the compile and link. Then, only a simple rule for the executable is necessary using the appropriate PETSc library variable. Individual compilation can be customized with the variables shown at the top of the makefile.

Users with large existing build systems may choose not to inherit the PETSc make rules but instead use a lower-level interface based only on PETSc make variables. A boilerplate example is given in Fig. 4. The user now includes only the `bmake/common/variables` file, which defines the make variables but does not prescribe any rules for compilation or linking. The variables provide information about all compilation flags, libraries, and external packages necessary to link with PETSc.

Before any PETSc code can be run, the user must call `PetscInitialize` (), and likewise after all PETSc code has completed the user must call `PetscFinalize` (). This is analogous to the requirements for using MPI. In fact, if the user has not done so already, PETSc will handle the initialization and cleanup of MPI automatically in these routines. A simple C driver is shown in Fig. 5.

The initialization call provides the command line arguments to PETSc for processing and can take an optional help string for the user (printed when the `-help` option is given). The finalization call frees any resources used by PETSc and provides

```
CFLAGS =
FFLAGS =
CPPFLAGS =
FPPFLAGS =

include ${PETSC_DIR}/bmake/common/base

ex1: ex1.o util.o chkopts
	-${CLINKER} -o $@ $^ ${PETSC_SNES_LIB}
	${RM} $^

ex1f: ex1f.o phys.o chkopts
	-${FLINKER} -o $@ $^ ${PETSC_FORTRAN_LIB} ${PETSC_SNES_LIB}
	${RM} $^
```

Fig. 3. Typical PETSc makefile.

```
include ${PETSC_DIR}/bmake/common/variables

.c.o:
	massageC.pl $<
	${CC} -c ${MY_CFLAGS} ${COPTFLAGS} ${CFLAGS} ${CCPPFLAGS} $<

.F.o:
	massageFortran.pl $<
	${FC} -c ${MY_FFLAGS} ${FOPTFLAGS} ${FFLAGS} ${FCPPFLAGS} $<

ex1: ex1.o util.o
	-${CLINKER} -o $@ $^ ${PETSC_SNES_LIB}
	${RM} $<

ex1f: ex1f.o phys.o
	-${FLINKER} -o $@ $^ ${PETSC_FORTRAN_LIB} ${PETSC_SNES_LIB}
	${RM} $<
```

Fig. 4. Boilerplate custom makefile using PETSc.

summary logging and diagnostic information, most notably performance profiling. The equivalent Fortran driver is shown in Fig. 6.

Notice that the command line arguments are now obtained directly from the Fortran runtime library, rather than the user code. Once these calls are inserted into the application code, the user can verify a successful link and run with the PETSc libraries.

```
static char help[] = "Boilerplate PETSc Example.\n\n";

#include "petsc.h"

int main(int argc, char **argv)
{
  ierr = PetscInitialize(&argc, &argv, (char *) 0, help);CHKERRQ(ierr);

  /* User code */

  return PetscFinalize();
}
```

Fig. 5. Boilerplate C driver for PETSc.

```
      program main
      implicit none
#include "include/finclude/petsc.h"

      integer ierr

      call PetscInitialize(PETSC_NULL_CHARACTER, ierr)

!     User code

      call PetscFinalize(ierr)
      end
```

Fig. 6. Boilerplate Fortran driver for PETSc.

3 Data Distribution and Linear Algebra

The PETSc solvers (discussed in Section 4) require the user to provide C or Fortran routines that compute the residual of the equation they wish to solve (after they have discretized the PDE) and optionally the Jacobian of that residual function. We refer to these functions generically as `FormFunction()` and `FormJacobian()`. For nonlinear problems the PETSc solvers use truncated Newton methods, with line searches or trust-regions for robustness, and possibly approximate or incomplete Jacobians for efficiency.

The central objects in any PETSc simulation are the abstractions from linear algebra, vectors and matrices, or in PETSc terms the `Vec` and `Mat` classes. These form the basis of all solver and preconditioner interfaces, as well as providing the link to user-supplied discretization and physics routines. Thus, the most important step in the migration of an application to the PETSc framework is the incorporation of its

linear algebra and data distribution abstractions. Two obvious strategies emerge for accomplishing this integration: a gradual approach that seeks to minimize the change to existing code and a more aggressive approach that leverages as much of the PETSc technology as possible. The next two subsections address these complementary paths toward integration. We use the simple example of the solid-fuel ignition problem, or Bratu problem, to illustrate the gradual evolution of a simulation, and then we give an example of mantle subduction for the more aggressive approach.

3.1 Gradual Evolution

A new PETSc user with very complicated code, which perhaps was originally written by someone else, may opt to make as few changes as possible when moving to PETSc. PETSc facilitates this approach with low-level interfaces to both `Vec` and `Mat` objects that closely resemble common serial data structures. Vectors are especially easy to migrate because the default PETSc storage format, contiguous arrays on each process, is that most common in serial applications. Matrices are somewhat more complicated, and in order to hide the actual matrix data storage format, PETSc requires the user to access values through a functional interface [6].

Let us examine the Bratu problem as an example of integrating PETSc vectors into an existing simulation [2]. This problem is modeled by the partial differential equation

$$-\Delta u - \lambda e^u = 0. \tag{4}$$

We take the domain to be the unit square and impose homogeneous Dirichlet boundary conditions on the edges. We discretize the equation using a 5-point stencil finite difference scheme, which results in a set of nonlinear algebraic equations $F(u_h) = 0$, where we use u_h to indicate the discrete solution vector. In Fig. 7, we present a Fortran 90 routine that calculates the residual F as a function of the input vector u_h. It also takes a user context as input that holds the domain information and problem coefficient.

Notice that the boundary conditions on the residual are of the form $u - u_\Gamma$, where u_Γ are the specified boundary values, because we are driving the residual $F(u)$ to zero. An alternative would be to eliminate those variables altogether, substituting the boundary values in any interior calculation. However, this technique currently imposes a greater burden on the programmer.

When integrating PETSc vectors into this code, we can let Fortran allocate the storage, or we can use PETSc for allocation. In Fig. 8, we let PETSc manage the memory. The input vectors already have storage allocated by PETSc, which can be accessed as an F90 array by using the `VecGetArrayF90()` function or in C by using `VecGetArray()`. A sample `driver()` routine shows how PETSc allocates vector storage. If we let Fortran manage memory, then we use `VecCreateSeqWithArray()`, or `VecCreateMPIWithArray()` in parallel, to create the `Vec` objects.

During a Newton iteration to solve this equation, we would require the Jacobian J of the mapping F. Rather than being accessed as a multidimensional array, the

```
      module f90module
      type userctx
!       The start, end, and number of vertices in the x- and y-directions
        integer xs,xe,ys,ye,integer mx,my
        double precision lambda
      end type userctx
      contains
      end module f90module

      subroutine FormFunction(u,F,user,ierr)
      use f90module
      type (userctx) user
      double precision u(user%xs:user%xe,user%ys:user%ye)
      double precision F(user%xs:user%xe,user%ys:user%ye)
      double precision two,one,hx,hy,hxdhy,hydhx,sc,uij,uxx,uyy
      integer i,j,ierr

      hx     = 1.0/dble(user%mx-1)
      hy     = 1.0/dble(user%my-1)
      sc     = hx*hy*user%lambda
      hxdhy  = hx/hy
      hydhx  = hy/hx

      do 20 j=user%ys,user%ye
         do 10 i=user%xs,user%xe
!           Apply boundary conditions
            if (i == 1 .or. j == 1 .or. i == user%mx .or. j == user%my) then
               F(i,j) = u(i,j)
!           Apply finite difference scheme
            else
               uij = u(i,j)
               uxx = hydhx * (2.0*uij - u(i-1,j) - u(i+1,j))
               uyy = hxdhy * (2.0*uij - u(i,j-1) - u(i,j+1))
               F(i,j) = uxx + uyy - sc*exp(uij)
            endif
 10      continue
 20   continue
```

Fig. 7. Residual calculation for the Bratu problem.

```
subroutine driver(u,F,user,ierr)
implicit none

Vec u,F
int N
parameter(N=10000)

call VecCreate(PETSC_COMM_WORLD,u,ierr)
call VecSetSizes(u,PETSC_DECIDE,N,ierr)
call VecDuplicate(u,F,ierr)
call SolverLoop(u,F,ierr)
call VecDestroy(u,ierr)
call VecDestroy(F,ierr)
end

subroutine FormFunctionPETSc(u,F,user,ierr)
use f90module
implicit none

Vec u,F
type (userctx) user
double precision,pointer :: u_v(:),f_v(:)

call VecGetArrayF90(u,u_v,ierr)
call VecGetArrayF90(F,f_v,ierr)
call FormFunction(u_v,f_v,user,ierr)
call VecRestoreArrayF90(u,u_v,ierr)
call VecRestoreArrayF90(F,f_v,ierr)
end
```

Fig. 8. Driver for the Bratu problem using PETSc.

PETSc `Mat` object provides the `MatSetValues` () function to set logically dense blocks of values into the structure. A function that computes the Jacobian and stores it in a PETSc matrix is shown in Fig. 9. With this addition, the user can now run serial code using the PETSc solvers (details are given in Section 4). Note that the storage mechanism is the only change to user code necessary for its incorporation into the PETSc framework using the wrapper in Fig. 8.

Although PETSc removes the need for low-level parallel programming, such as direct calls to the MPI library, the user must still identify parallelism inherent in the application and must structure the computation accordingly. The first step is to partition the domain and have each process calculate the residual and Jacobian only over its local piece. This is easily accomplished by redefining the `xs`, `xe`, `ys`, and `ye` variables on each process, converting loops over the entire domain into loops over the local domain. However, this method uses nearest-neighbor information, and thus

```
      subroutine FormJacobian(u,jac,user,ierr)
      use f90module
      type (userctx) user
      Mat jac
      double precision x(user%xs:user%xe,user%ys:user%ye)
      double precision two,one,hx,hy,hxdhy,hydhx,sc,v(5)
      integer row,col(5),i,j,ierr

      one    = 1.0
      two    = 2.0
      hx     = one/dble(user%mx-1)
      hy     = one/dble(user%my-1)
      sc     = hx*hy*user%lambda
      hxdhy  = hx/hy
      hydhx  = hy/hx
      do 20 j=user%ys,user%ye
         row = (j - user%ys)*user%xm - 1
         do 10 i=user%xs,user%xe
            row = row + 1
!           boundary points
            if (i == 1 .or. j == 1 .or. i == user%mx .or. j == user%my) then
               col(1) = row
               v(1)   = one
               call MatSetValues(jac,1,row,1,col,v,INSERT_VALUES,ierr)
!           interior grid points
            else
               v(1) = -hxdhy
               v(2) = -hydhx
               v(3) = two*(hydhx + hxdhy) - sc*exp(x(i,j))
               v(4) = -hydhx
               v(5) = -hxdhy
               col(1) = row - user%xm
               col(2) = row - 1
               col(3) = row
               col(4) = row + 1
               col(5) = row + user%xm
               call MatSetValues(jac,1,row,5,col,v,INSERT_VALUES,ierr)
            endif
 10      continue
 20   continue
```

Fig. 9. Jacobian calculation for the Bratu problem using PETSc.

we must have access to some values not present locally on the process, as shown in Fig. 10.

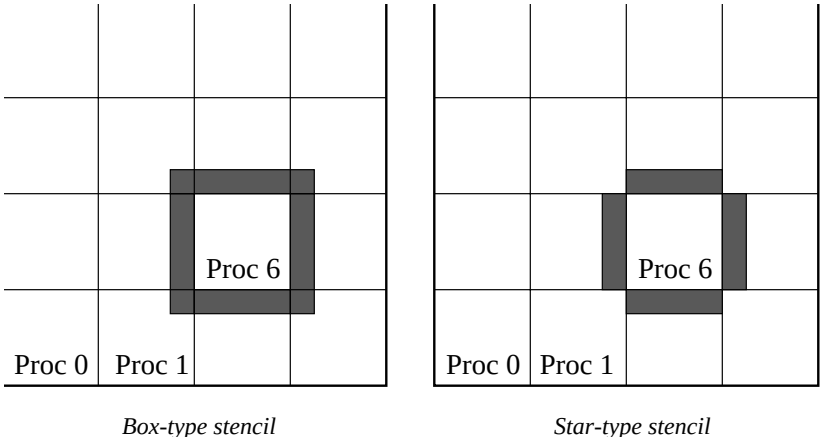

Fig. 10. Star and box stencils using a DA.

We must store values for these *ghost* points locally so that they may be used for the computation. In addition, we must ensure that these values are identical to the corresponding values on the neighboring processes. At the lowest level, PETSc provides *ghosted vectors*, created using `VecCreateGhost` (), with storage for some values owned by other processes. These values are explicitly indicated during construction. The `VecGhostUpdateBegin` () and `VecGhostUpdateEnd` () functions transfer values between ghost storage and the corresponding storage on other processes. This method, however, can become complicated for the user. Therefore, for the common case of a logically rectangular grid, PETSc provides the `DA` object to manage the determination, allocation, and coherence of ghost values. The `DA` object is discussed fully in Section 3.2, but we give here a short introduction in the context of the Bratu problem.

The `DA` object represents a distributed, structured (possibly staggered) grid and thus has more information than our serial grid. If we augment our user context to include information about ghost values, we can obtain all the grid information from the `DA` . Figure 11 shows the creation of a `DA` when one initially know only the total number of vertices in the x and y directions. Alternatively, we could have specified the local number of vertices in each direction.

Now we need only modify our PETSc residual routine to update the ghost values, as shown in Fig. 12.

Notice that the ghost value scatter is a two-step operation. This allows the communication to overlap with local computation that may be taking place while the messages are in transit. The only change necessary for the residual calculation itself is the correct declaration of the input array, `double precision`

```
      type userctx
        DA da
!       The start, end, and number of vertices in the x−direction
        integer xs,xe,xm
!       The start, end, and number of ghost vertices in the x−direction
        integer gxs,gxe,gxm
!       The start, end, and number of vertices in the y−direction
        integer ys,ye,ym
!       The start, end, and number of ghost vertices in the y−direction
        integer gys,gye,gym
!       The number of vertices in the x− and y−directions
        integer mx,my
!       The MPI rank of this process
        integer rank
!       The coefficient in the Bratu equation
        double precision lambda
      end type userctx

      call DACreate2d(PETSC_COMM_WORLD,DA_NONPERIODIC,DA_STENCIL_STAR, &
     &      user%mx,user%my,PETSC_DECIDE,PETSC_DECIDE,1,1,             &
     &      PETSC_NULL_INTEGER,PETSC_NULL_INTEGER,user%da,ierr)
      call DAGetCorners(user%da,user%xs,user%ys,PETSC_NULL_INTEGER,  &
     &      user%xm,user%ym,PETSC_NULL_INTEGER,ierr)
      call DAGetGhostCorners(user%da,user%gxs,user%gys,              &
     &      PETSC_NULL_INTEGER,user%gxm,user%gym,                    &
     &      PETSC_NULL_INTEGER,ierr)
!   Here we shift the starting indices up by one so that we can easily
!   use the Fortran convention of 1−based indices, rather than 0−based.
      user%xs  = user%xs+1
      user%ys  = user%ys+1
      user%gxs = user%gxs+1
      user%gys = user%gys+1
      user%ye  = user%ys+user%ym−1
      user%xe  = user%xs+user%xm−1
      user%gye = user%gys+user%gym−1
      user%gxe = user%gxs+user%gxm−1
```

Fig. 11. Creation of a DA for the Bratu problem.

```
      subroutine FormFunctionPETSc(u,F,user,ierr)
      implicit none

      Vec u,F
      integer ierr
      type (userctx) user

      double precision,pointer :: lu_v(:),lf_v(:)
      Vec uLocal

      call DAGetLocalVector(user%da,uLocal,ierr)
      call DAGlobalToLocalBegin(user%da,u,INSERT_VALUES,uLocal,ierr)
      call DAGlobalToLocalEnd(user%da,u,INSERT_VALUES,uLocal,ierr)

      call VecGetArrayF90(uLocal,lu_v,ierr)
      call VecGetArrayF90(F,lf_v,ierr)

!     Actually compute the local portion of the residual
      call FormFunctionLocal(lu_v,lf_v,user,ierr)

      call VecRestoreArrayF90(uLocal,lu_v,ierr)
      call VecRestoreArrayF90(F,lf_v,ierr)

      call DARestoreLocalVector(user%da,uLocal,ierr)
```

Fig. 12. Parallel residual calculation for the Bratu problem using PETSc.

`u(user%gxs:user%gxe,user%gys:user%gye)`, which now includes ghost values,

The changes to `FormJacobian()` are similar, resizing the input array and changing slightly the calculation of row and column indices, and can be found in the PETSc example source. We have now finished the initial introduction of PETSc linear algebra, and the simulation is ready to run in parallel.

3.2 Rapid Evolution

The user who is willing to raise the level of abstraction in a code can start by employing the `DA` to describe the problem domain and discretization. The data interface mimics a multidimensional array and is thus ideal for finite difference, finite volume, and low-order finite element schemes on a logically rectangular grid. In fact, after the definition of a `FormFunction()` routine, the simulation is ready to run because the nonlinear solver provides approximate Jacobians automatically, as discussed in Section 4.

```
DASetLocalFunction(user.da, (DALocalFunction1) FormFunctionLocal);
```

Fig. 13. Using the DA to form a function.

```
int FormFunctionLocal(DALocalInfo *info,double **u,double **f,AppCtx *user){
  double  two = 2.0,hx,hy,hxdhy,hydhx,sc;
  double  uij,uxx,uyy;
  int     i,j;

  hx     = 1.0/(double)(info->mx-1);
  hy     = 1.0/(double)(info->my-1);
  sc     = hx*hy*user->lambda;
  hxdhy  = hx/hy;
  hydhx  = hy/hx;
  /* Compute function over the locally owned part of the grid */
  for (j=info->ys; j<info->ys+info->ym; j++) {
    for (i=info->xs; i<info->xs+info->xm; i++) {
      if (i == 0 || j == 0 || i == info->mx-1 || j == info->my-1) {
        f[j][i] = u[j][i];
      } else {
        uij     = u[j][i];
        uxx     = (two*uij - u[j][i-1] - u[j][i+1])*hydhx;
        uyy     = (two*uij - u[j-1][i] - u[j+1][i])*hxdhy;
        f[j][i] = uxx + uyy - sc*exp(uij);
      }
    }
  }
}
```

Fig. 14. `FormFunctionLocal()` for the Bratu problem.

We begin by examining the Bratu problem from the last section, only this time in C. We can now provide our local residual routine to the `DA` ; see Fig. 13.

The grid information is passed to `FormFunctionLocal()` in a `DALocalInfo` structure, as shown in Fig. 14.

The fields are passed directly as multidimensional C arrays, complete with ghost values. An application scientist can use a boilerplate example, such as the Bratu problem, merely altering the local physics calculation in `FormFunctionLocal()`, to rapidly obtain a parallel, scalable simulation that runs on the desktop as well as on massively parallel supercomputers.

If the user has an expression for the Jacobian of the residual function, then this matrix can also be computed by using the `DA` interface with the call, as in Fig. 15, and the corresponding routine for the local calculation in Fig. 16.

```
DASetLocalJacobian(user.da, (DALocalFunction1) FormJacobianLocal);
```

Fig. 15. Using the DA to form a Jacobian.

```
int FormJacobianLocal(DALocalInfo *info,double **x,Mat jac,AppCtx *user){
  MatStencil    col[5],row;
  double        lambda,v[5],hx,hy,hxdhy,hydhx,sc;
  int           i,j;

  lambda = user->param;
  hx     = 1.0/(double)(info->mx-1);
  hy     = 1.0/(double)(info->my-1);
  sc     = hx*hy*lambda;
  hxdhy  = hx/hy;                          hydhx  = hy/hx;

  for (j=info->ys; j<info->ys+info->ym; j++) {
    for (i=info->xs; i<info->xs+info->xm; i++) {
      row.j = j; row.i = i;
      /* boundary points */
      if (i == 0 || j == 0 || i == info->mx-1 || j == info->my-1) {
        v[0] = 1.0;
        MatSetValuesStencil(jac,1,&row,1,&row,v,INSERT_VALUES);
      } else {
      /* interior grid points */
        v[0] = -hxdhy; col[0].j = j - 1; col[0].i = i;
        v[1] = -hydhx; col[1].j = j; col[1].i = i-1;
        v[2] = 2.0*(hydhx + hxdhy) - sc*exp(x[j][i]); col[2].j = j; col[2].i = i;
        v[3] = -hydhx; col[3].j = j; col[3].i = i+1;
        v[4] = -hxdhy; col[4].j = j + 1; col[4].i = i;
        MatSetValuesStencil(jac,1,&row,5,col,v,INSERT_VALUES);
      }
    }
  }
  MatAssemblyBegin(jac,MAT_FINAL_ASSEMBLY);
  MatAssemblyEnd(jac,MAT_FINAL_ASSEMBLY);
  MatSetOption(jac,MAT_NEW_NONZERO_LOCATION_ERR);
}
```

Fig. 16. `FormJacobianLocal()` for the Bratu problem.

```
int FormFunctionLocal(DALocalInfo *info,Field **x,Field **f,void *ptr){
  AppCtx      *user = (AppCtx*)ptr;
  Parameter   *param = user->param;
  GridInfo    *grid  = user->grid;
  double      mag_w, mag_u;
  int         ilim = info->mx-1, jlim = info->my-1, i, j;

  for (j=info->ys; j<info->ys+info->ym; j++) {
    for (i=info->xs; i<info->xs+info->xm; i++) {
      calculateXMomentumResidual(i, j, x, f, user);
      calculateZMomentumResidual(i, j, x, f, user);
      calculatePressureResidual(i, j, x, f, user);
      calculateTemperatureResidual(i, j, x, f, user);
    }
  }
}
```

Fig. 17. `FormFunctionLocal()` for the mantle subduction problem.

We have so far presented simple examples involving a perturbed Laplacian, but this development strategy extends far beyond toy problems. We have developed large-scale, parallel geophysical simulations [1] that are producing new results in the field. We began by replacing `FormFunctionLocal()` in the Bratu example with one containing the linear physics of the previous sequential linear subduction code (dropping the previous code's linear solve). Then the PETSc solver results were compared, for correctness, with the complete prior subduction code. Next the new code was immediately run in parallel for correctness and efficiency tests. Finally the nonlinear terms from the more complicated physics of the mantle subduction problem, discussed in Section 1 and shown in Figs. 17-21 and better quality discretizations of the boundary conditions were added. In fact, since the structure of the `FormFunction()` changed significantly, the final `FormFunction()` looks nothing like the original, but the process of obtaining through the incremental approach reduced the learning curve for PETSc and make correctness checking at the various stages much easier.

The details of the subduction code are not as important as the recognition that a very complicated physics problem need not involve any more complication in our simulation infrastructure. To give more insight into the actual calculation, we show in Fig. 22 the explicit calculation of the residual from the continuity equation.

Here we have used the `DA` for a multicomponent problem on a staggered mesh without alteration because the structure of the discretization remains logically rectangular.

Although the `DA` manages communication during the solution process, data postprocessing, often necessary for visualization or analysis, also requires this functionality. For the mantle subduction simulation, we want to calculate both the second in-

```
int calculateXMomentumResidual(int i, int j, Field **x,Field **f,AppCtx *user){
  Parameter *param = user->param;
  GridInfo   *grid   = user->grid;
  int         ilim = info->mx-1, jlim = info->my-1;

  if (i<j) {
    f[j][i].u = x[j][i].u - SlabVel('U',i,j,user);
  } else if (j<=grid->jlid || (j<grid->corner+grid->inose && i<grid->corner+grid->inose)) {
    /* in the lithospheric lid */
    f[j][i].u = x[j][i].u - 0.0;
  } else if (i==ilim) { /* on the right side boundary */
    if (param->ibound==BC_ANALYTIC) {
      f[j][i].u = x[j][i].u - HorizVelocity(i,j,user);
    } else {
      f[j][i].u = XNormalStress(x,i,j,CELL_CENTER,user) - EPS_ZERO;
    }
  } else if (j==jlim) { /* on the bottom boundary */
    if (param->ibound==BC_ANALYTIC) {
      f[j][i].u = x[j][i].u - HorizVelocity(i,j,user);
    } else if (param->ibound==BC_NOSTRESS) {
      f[j][i].u = XMomentumResidual(x,i,j,user);
    }
  } else { /* in the mantle wedge */
    f[j][i].u = XMomentumResidual(x,i,j,user);
  }
}
```

Fig. 18. X-momentum residual for the mantle subduction problem.

variant of the strain rate tensor and the viscosity over the grid (at both cell centers and corners). Using the `DAGlobalToLocalBegin` () and `DAGlobalToLocalEnd` () functions, we transferred the global solution data to local ghosted vectors and proceeded with the calculation. We can also transfer data from local vectors to a global vector using `DALocalToGlobal` (). The entire viscosity calculation is given in Fig. 23.

Note that this framework allows the user to manage arbitrary fields over the domain. For instance, a porous flow simulation might manage material properties of the medium in this fashion.

4 Solvers

The `SNES` object in PETSc is an abstraction of the "inverse" of a nonlinear operator. The user provides a `FormFunction()` routine, as seen in Section 3, which computes the action of the operator on an input vector. Linear problems can also be

```
int calculateZMomentumResidual(int i, int j, Field **x,Field **f,AppCtx *user){
  Parameter *param = user->param;
  GridInfo   *grid   = user->grid;
  int         ilim = info->mx-1,jlim = info->my-1;

  if (i<=j) {
    f[j][i].w = x[j][i].w - SlabVel('W',i,j,user);
  } else if (j<=grid->jlid || (j<grid->corner+grid->inose && i<grid->corner+grid->inose)) {
    /* in the lithospheric lid */
    f[j][i].w = x[j][i].w - 0.0;
  } else if (j==jlim) { /* on the bottom boundary */
    if (param->ibound==BC_ANALYTIC) {
      f[j][i].w = x[j][i].w - VertVelocity(i,j,user);
    } else {
      f[j][i].w = ZNormalStress(x,i,j,CELL_CENTER,user) - EPS_ZERO;
    }
  } else if (i==ilim) { /* on the right side boundary */
    if (param->ibound==BC_ANALYTIC) {
      f[j][i].w = x[j][i].w - VertVelocity(i,j,user);
    } else if (param->ibound==BC_NOSTRESS) {
      f[j][i].w = ZMomentumResidual(x,i,j,user);
    }
  } else { /* in the mantle wedge */
    f[j][i].w =      ZMomentumResidual(x,i,j,user);
  }
}
```

Fig. 19. Z-momentum residual for the mantle subduction problem.

```
int calculatePressureResidual(int i, int j, Field **x,Field **f,AppCtx *user){
  Parameter *param = user->param;
  GridInfo   *grid   = user->grid;
  int         ilim = info->mx-1, jlim = info->my-1;

  if (i<j || j<=grid->jlid || (j<grid->corner+grid->inose && i<grid->corner+grid->inose)) {
    /* in the lid or slab */
    f[j][i].p = x[j][i].p;
  } else if ((i==ilim || j==jlim) && param->ibound==BC_ANALYTIC) { /* on an analytic boundary */
    f[j][i].p = x[j][i].p - Pressure(i,j,user);
  } else { /* in the mantle wedge */
    f[j][i].p = ContinuityResidual(x,i,j,user);
  }
}
```

Fig. 20. Pressure, or continuity, residual for the mantle subduction problem.

```
int calculateTemperatureResidual(int i, int j, Field **x,Field **f,AppCtx *user){
  Parameter *param = user->param;
  GridInfo   *grid   = user->grid;
  int         ilim = info->mx-1, jlim = info->my-1;

  if (j==0) { /* on the surface */
    f[j][i].T = x[j][i].T + x[j+1][i].T + PetscMax(x[j][i].T,0.0);
  } else if (i==0) { /* slab inflow boundary */
    f[j][i].T = x[j][i].T - PlateModel(j,PLATE_SLAB,user);
  } else if (i==ilim) { /* right side boundary */
    mag_u = 1.0 - pow( (1.0-PetscMax(PetscMin(x[j][i-1].u/param->cb,1.0),0.0)), 5.0 );
    f[j][i].T = x[j][i].T - mag_u*x[j-1][i-1].T - (1.0-mag_u)*PlateModel(j,PLATE_LID,user);
  } else if (j==jlim) { /* bottom boundary */
    mag_w = 1.0 - pow( (1.0-PetscMax(PetscMin(x[j-1][i].w/param->sb,1.0),0.0)), 5.0 );
    f[j][i].T = x[j][i].T - mag_w*x[j-1][i-1].T - (1.0-mag_w);
  } else { /* in the mantle wedge */
    f[j][i].T = EnergyResidual(x,i,j,user);
  }
}
```

Fig. 21. Temperature, or energy, residual for the mantle subduction problem.

```
double precision ContinuityResidual(Field **x, int i, int j, AppCtx *user)
{
  GridInfo             *grid = user->grid;
  double precision uE,uW,wN,wS,dudx,dwdz;

  uW = x[j][i-1].u; uE = x[j][i].u; dudx = (uE - uW)/grid->dx;
  wS = x[j-1][i].w; wN = x[j][i].w; dwdz = (wN - wS)/grid->dz;
  return dudx + dwdz;
}
```

Fig. 22. Calculation of the continuity residual for the mantle subduction problem.

solved by using SNES with no loss of efficiency compared to using the underlying KSP object (PETSc linear solver object) directly. Thus SNES seems the appropriate framework for a general-purpose simulator, providing the flexibility to add or subtract nonlinear terms from the equation at will.

The SNES solver does not require a user to implement a Jacobian. Default routines are provided to compute a finite difference approximation using coloring to account for the sparsity of the matrix. The user does not even need the PETSc Mat interface but can initially interact only with Vec objects, making the transition to PETSc nearly painless. However, these approximations to the Jacobian can have numerical difficulties and are not as efficient as direct evaluation. Therefore, the user

```
/* Compute both the second invariant of the strain rate tensor and the viscosity */
int ViscosityField(DA da, Vec X, Vec V,AppCtx *user){
  Parameter     *param = user->param;
  GridInfo      *grid  = user->grid;
  Vec           localX;
  Field         **v, **x;
  double        eps, dx, dz, T, epsC, TC;
  int           i,j,is,js,im,jm,ilim,jlim,ivt;

  ivt          = param->ivisc;
  param->ivisc = param->output_ivisc;

  DACreateLocalVector(da, &localX);
  DAGlobalToLocalBegin(da, X, INSERT_VALUES, localX);
  DAGlobalToLocalEnd(da, X, INSERT_VALUES, localX);
  DAVecGetArray(da,localX,(void**)&x);
  DAVecGetArray(da,V,(void**)&v);

  /* Parameters */
  dx   = grid->dx;   dz   = grid->dz;
  ilim = grid->ni-1; jlim = grid->nj-1;

  /* Compute real temperature, strain rate and viscosity */
  DAGetCorners(da,&is,&js,PETSC_NULL,&im,&jm,PETSC_NULL);
  for (j=js; j<js+jm; j++) {
    for (i=is; i<is+im; i++) {
      T = param->potentialT * x[j][i].T * exp( (j-0.5)*dz*param->z_scale );
      if (i<ilim && j<jlim) {
        TC = param->potentialT * TInterp(x,i,j) * exp( j*dz*param->z_scale );
      } else {
        TC = T;
      }
      /* Compute the values at both cell centers and cell corners */
      eps  = CalcSecInv(x,i,j,CELL_CENTER,user);
      epsC = CalcSecInv(x,i,j,CELL_CORNER,user);
      v[j][i].u = eps;
      v[j][i].w = epsC;
      v[j][i].p = Viscosity(T,eps,dz*(j-0.5),param);
      v[j][i].T = Viscosity(TC,epsC,dz*j,param);
    }
  }
  DAVecRestoreArray(da,V,(void**)&v);
  DAVecRestoreArray(da,localX,(void**)&x);
  param->ivisc = ivt;
}
```

Fig. 23. Calculation of the second invariant of the strain tensor and viscosity field for the mantle subduction problem.

```
KSP ksp, *subksp;
PC bjacobi, ilu;

SNESGetKSP(snes, &ksp);
KSPGetPC(ksp, &bjacobi);
PCBJacobiGetSubKSP(bjacobi, PETSC_NULL, PETSC_NULL, &subksp);
KSPGetPC(subksp[0], &ilu);
PCILUSetLevels(ilu, levels);
```

Fig. 24. Customizing the preconditioner on a single process.

has the option of providing a routine or of using the ADIC [15] or similar system for automatic differentiation. The finite difference approximation can be activated by using the `-snes_fd` and `-mat_fd_coloring_freq` options or by providing the `SNESDefaultComputeJacobianColor` () function to `SNESSetJacobian` ().

Through the `SNES` object, the user may also access the great range of direct and iterative linear solvers and preconditioners provided by PETSc. Sixteen different Krylov solvers are available including GMRES, BiCGStab, and LSQR, along with interfaces to popular sparse direct packages, such as MUMPS [4] and SuperLU [9]. In addition to a variety of incomplete factorization preconditioners, including PILUT [16], PETSc supports additive Schwartz preconditioning, algebraic multigrid through BoomerAMG [13], and the geometric multigrid discussed below. The full panoply of solvers and preconditioners available is catalogued on the PETSc Website [3].

The modularity of PETSc allows users to easily customize each solver. For instance, suppose the user wishes to increase the number of levels in ILU(k) preconditioning on one block of a block-Jacobi scheme. The code fragment in Fig. 24 will set the number of levels of fill to `levels`.

PETSc provides an elegant framework for managing geometric multigrid in combination with a `DA` , which is abstracted in the `DMMG` object. In the same way that any linear solve can be performed with `SNES` , any preconditioner or solver combination is available in `DMMG` by using a single level. That is, the same user code supports both geometric multigrid as well as direct methods, Krylov methods etc. When creating a `DMMG` , the user specifies the number of levels of grid refinement and provides the coarse-grid information (see Fig. 25), which is always a `DA` object at present, although unstructured prototypes are being developed. Then the `DMMG` object calculates the intergrid transfer operators, prolongation and restriction, and allocates objects to hold the solution at each level. The user must provide the action of the residual operator F and can optionally provide a routine to compute the Jacobian matrix (which will be called on each level) and a routine to compute the initial guess, as in Fig. 26.

With this information, the user can now solve the equation and retrieve the solution vector, as in Fig. 27.

```
DMMGCreate(comm, grid.mglevels, user, &dmmg);
DMMGSetDM(dmmg, (DM) da);
```

Fig. 25. Creating a DMMG.

```
DMMGSetSNESLocal(dmmg, FormFunctionLocal, FormJacobianLocal, 0, 0);
DMMGSetInitialGuess(dmmg, FormInitialGuess);
```

Fig. 26. Providing discretization and assembly routines to DMMG.

```
DMMGSolve(dmmg);
soln = DMMGGetx(dmmg);
```

Fig. 27. Solving the problem with DMMG.

5 Extensions

PETSc is not a complete environment for simulating physical phenomena. Rather it is a set of tools that allow the user to assemble such an environment tailored to a specific application. As such, PETSc will never provide every facility appropriate for a given simulation. However, the user can easily extend PETSc and supplement its capabilities. We present two examples in the context of the mantle subduction simulation.

5.1 Simple Continuation

Because of the highly nonlinear dependence of viscosity on temperature and strain rate in equation (1), the iteration in Newton's method can fail to converge without a good starting guess of the solution. On the other hand, for the isoviscous case where temperature is coupled to velocity only through advection, a solution is easily reached. It is therefore natural to propose a continuation scheme in the viscosity beginning with constant viscosity and progressing to full variability. A simple adaptive continuation model was devised, in which the viscosity was raised to a power between zero and one, $\eta \rightarrow \eta^{\alpha}$, where α=0 corresponds to constant viscosity and α=1 corresponds to natural viscosity variation. In the continuation loop, α was increased towards unity by an amount dependent on the rate of convergence of the previous `SNES` solve.

PETSc itself provides no special support for continuation, but it is sufficiently modular that a continuation loop was readily constructed in the application code, using repeated calls to `DMMGSolve` (), and found to work quite well.

5.2 Simple Steering

No rigorous justification had been given for the continuation strategy above, and consequently it was possible that, for certain initial conditions, Newton would fail to converge, thereby resulting in extremely long run times. An observant user could detect this situation and abort the run, but all the potentially useful solution information up to that point would be lost. A clean way to asynchronously abort the continuation loop was thus needed. On architectures where OS signals are available, PETSc provides an interface for registering signal handlers. Thus we were able to define a handler. The user wishing to change the control flow of the simulation simply sends the appropriate signal to the process. This sets a flag in the user data that causes the change at the next iteration.

6 Simulation Results

A long-standing debate has existed between theorists and observationalists over the thermal structure of subduction zones. Modelers, using constant viscosity flow simulations to compute thermal structure, have predicted relatively cold subduction zones [19]. Conversely, observationalists, who use heat flow measurements and mineralogical thermobarometry to estimate temperatures at depth, have long claimed that subduction zones are hotter than predicted. They have invoked the presence of strong upwelling of hot mantle material to supply the heat. This upwelling was never predicted by isoviscous flow models, however.

The debate remained unresolved until recently when several groups, including our own, succeeded in developing simulations with realistically variable viscosity (for other examples [8, 10, 12, 18, 21]). A comparison of flow and temperature fields for variable and constant viscosity simulations generated with our code is shown in Fig. 28. Results from these simulations are exciting because they close the gap between models and observations: they predict hotter mantle temperatures, steeper surface thermal gradients, and upwelling mantle flow.

Furthermore, these simulations allow for quantitative predictions of variation of observables with subduction parameters. A recent study of subduction zone earthquakes has identified an intriguing trend: the vertical distance from subduction zone volcanoes to the surface of the subducting slab is anti-correlated with descent rate of the slab [11]. Preliminary calculations with our model are consistent with this trend, indicating that flow and thermal structure may play an important role in determining not only the quantity and chemistry of magmas but also their path of transport to the surface. Further work is required to resolve this issue.

Simulating the geodynamics of subduction is an example of our success in porting an existing code into the PETSc framework, simultaneously parallelizing the code and increasing its functionality. Using this code as a template, we rapidly developed a related simulation of another tectonic boundary, the mid-ocean ridge. This work was done to address a set of observations of systematic morphological asymmetry in the global mid-ocean ridge system [7]. Our model confirms the qualitative

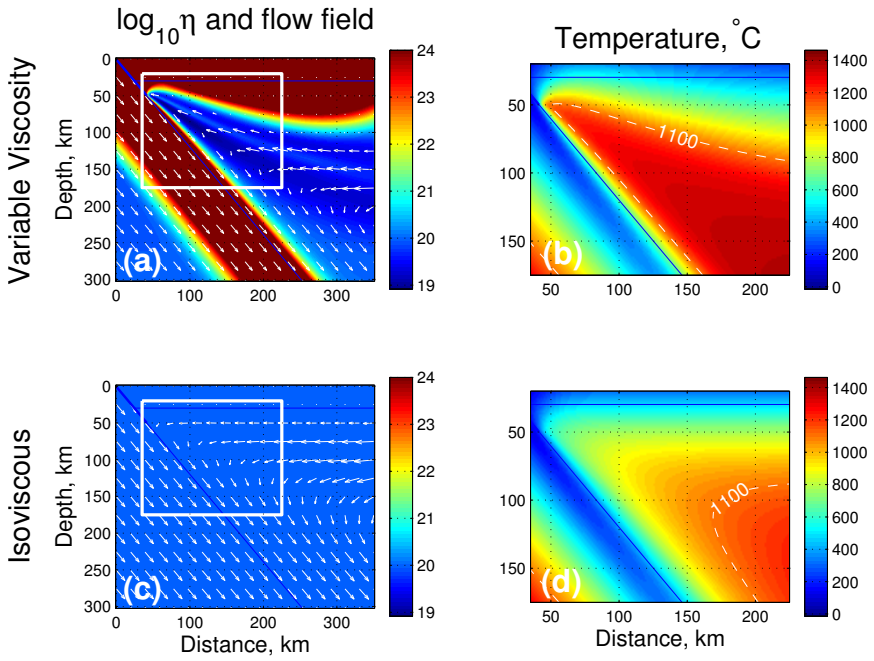

Fig. 28. 2D viscosity and potential temperature fields from simulations on 8 processors with 230,112 degrees of freedom. Panels in the top row are from a simulation with α=1 in equation (1). Panels in bottom row have α=0. The white box in panels (a) and (c) shows the region in which temperature is plotted in panels (b) and (d). **(a)** Colors show $\log_{10}$ of the viscosity field. Note that there are more than five orders of magnitude variation in viscosity. Arrows show the flow direction and magnitude (the slab is being subducted at a rate of 6 cm/year). Upwelling is evident in the flow field near the wedge corner. **(b)** Temperature field from the variable viscosity simulation; 1100°C isotherm is shown as a dashed line. **(c)** (Constant) viscosity and flow field from the isoviscous simulation. Strong flow is predicted at the base of the crust despite the low-temperature rock there. No upwelling flow is predicted. **(d)** Temperature field from isoviscous simulation. Note that the mantle wedge corner is much colder than in (b).

mechanism that had been proposed to explain these observations. Furthermore, it shows a quantitative agreement with trends in the observed data [17]. As with our models of subduction, the key to demonstrating the validity of the hypothesized dynamics was simulating them with the strongly nonlinear rheology that characterizes mantle rock on geologic timescales. The computational tools provided by PETSc enabled us easily handle this rheology, reducing development time and allowing us to focus on model interpretation.

References

1. PETSc SNES Example 30. `http://www.mcs.anl.gov/petsc/petsc-2/`

snapshots/petsc-current/src/snes/examples/tutorials/ex30.c.html
2. PETSc SNES Example 5. http://www.mcs.anl.gov/petsc/petsc-2/snapshots/petsc-current/src/snes/examples/tutorials/ex5f90.F.html
3. PETSc Solvers. http://www.mcs.anl.gov/petsc/petsc-2/documentation/linearsolvertable.html
4. Amestoy, P. R., Duff, I. S., L'Excellent, J.-Y., and Koster, J.: A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, (2001)
5. Balay, S., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H.: PETSc Web page. http://www.mcs.anl.gov/petsc
6. Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F.: Efficient management of parallelism in object oriented numerical software libraries. In Arge, E., Bruaset, A. M., and Langtangen, H. P., editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, (1997)
7. Carbotte, S., Small, C., and Donnelly, K.: The influence of ridge migration on the magmatic segmentation of mid-ocean ridges. *Nature*, 429:743–746, (2004)
8. Conder, J., Wiens, D., and Morris, J.: On the decompression melting structure at volcanic arcs and back-arc spreading centers. *Geophys. Res. Letts.*, 29, (2002)
9. Demmel, J. W., Gilbert, J. R., and Li, X. S.: SuperLU user's guide. Technical Report LBNL-44289, Lawrence Berkeley National Laboratory, (2003)
10. Eberle, M., Grasset, O., and Sotin, C.: A numerical study of the interaction of the mantle wedge, subducting slab, and overriding plate. *Phys. Earth Planet. In.*, 134:191–202, (2002)
11. England, P., Engdahl, R., and Thatcher, W.: Systematic variation in the depth of slabs beneath arc volcanos. *Geophys. J. Int.*, 156(2):377–408, (2003)
12. Furukawa, Y.: Depth of the decoupling plate interface and thermal structure under arcs. *J. Geophys. Res.*, 98:20005–20013, (1993)
13. Henson, V. E. and Yang, U. M.: BoomerAMG: A parallel algebraic multigrid solver and preconditioner. Technical Report UCRL-JC-133948, Lawrence Livermore National Laboratory, (2000)
14. Hirth, G. and Kohlstedt, D.: Rheology of the upper mantle and the mantle wedge: A view from the experimentalists. In *Inside the Subduction Factory*, volume 138 of *Geophysical Monograph*. American Geophysical Union, (2003)
15. Hovland, P., Norris, B., and Smith, B.: Making automatic differentiation truly automatic: Coupling PETSc with ADIC. In *Proceedings of ICCS2002*, (2002)
16. Hysom, D. and Pothen, A.: A scalable parallel algorithm for incomplete factor preconditioning. *SIAM Journal on Scientific Computing*, 22:2194–2215, (2001)
17. Katz, R., Spiegelman, M., and Carbotte, S.: Ridge migration, asthenospheric flow and the origin of magmatic segmentation in the global mid-ocean ridge system. *Geophys. Res. Letts.*, 31, (2004)
18. Kelemen, P., Rilling, J., Parmentier, E., Mehl, L., and Hacker, B.: Thermal structure due to solid-state flow in the mantle wedge beneath arcs. In *Inside the Subduction Factory*, volume 138 of *Geophysical Monograph*. American Geophysical Union, (2003)
19. Peacock, S. and Wang, K.: Seismic consequences of warm versus cool subduction metamorphism: Examples from southwest and northeast Japan. *Science*, 286:937–939, (1999)
20. Stern, R.: Subduction zones. *Rev. Geophys.*, 40(4), (2002)

21. van Keken, P., Kiefer, B., and Peacock, S.: High-resolution models of subduction zones: Implications for mineral dehydration reactions and the transport of water into the deep mantle. *Geochem. Geophys. Geosys.*, 3(10), (2003)