Mecánica Computacional Vol. XXIV
A. Larreteguy (Editor)
Buenos Aires, Argentina, Noviembre 2005

PARALLEL FEM APPLICATION DEVELOPMENT IN PYTHON

**Lisandro D. Dalcín, Rodrigo R. Paz, Andrés A. Anca,
Mario A. Storti and Jorge D'Elía**

Centro Internacional de Métodos Computacionales en Ingeniería (CIMEC)
CONICET - INTEC - UNL
Parque Tecnológico del Litoral Centro
(3000) Santa Fe, Argentina
e-mail: (dalcinl|rodrigop|aanca|mstorti|jdelia)@intec.unl.edu.ar

Key Words: Parallel Computing, FEM, High-level Languages, Python.

Abstract. *Python is a modern but mature, easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python codes are quickly developed, easily debugged and maintained, and can achieve a high degree of integration with other libraries written in compiled languages. Those characteristics make Python an ideal candidate for writing the higher-level parts of large-scale scientific applications and driving simulations in parallel architectures like clusters of PC's or SMP's.*

In this work, we present some parallel finite element simulations driven in a cluster of PC's using the Python programming language. Our previously developed MPI, PETSc *and* ParMETIS *packages are used together to deploy a model parallel FEM framework, integrating an important subset of* OOFELIE *toolkit, a sequential C++ code for FEM simulation and development. Our main concern is in showing the advantages of using high-level scripting languages for the high-level part of that kind of codes, where parallelization introduces some extra complexities.*

1 INTRODUCTION

Over the last years, high performance computing has become an affordable resource to many more researchers in the scientific community than ever before. The conjunction of quality open source software and commodity hardware strongly influenced the now widespread popularity of Beowulf[1] class clusters and cluster of workstations.

Among many parallel computational models, message-passing has proved to be an effective one. This paradigm is specially suited for (but not limited to) distributed memory architectures and is used in today's most demanding scientific and engineering application related to modeling, simulation, design, and signal processing. However, portable message-passing parallel programming used to be a nightmare in the past because of the many incompatible options developers were faced to. Fortunately, this situation definitely changed after the MPI Forum[2] released its standard specification.

High performance computing is traditionally associated with software development using compiled languages. However, in typical applications programs, only a small parts of codes are time-critical enough to require the efficiency of compiled languages. The other parts are generally related to memory management, error handling, input/output, and user interaction, and those are usually the most error prone and time-consuming lines of code to write and debug.

Interpreted languages can be really advantageous for implementing the high-level parts of any application. They are well established in the scientific community. In the commercial side, MATLAB is the dominant interpreted programming language for implementing general numerical computations. In the open source side, Octave and Scilab are well known, freely distributed software packages providing compatibility with MATLAB language. Even in a specialized application domain like multi-physics simulations by finite element methods, the developers of *OOFELIE*[3,4] toolkit had early realized the importance of providing end-users with a high-level, interpreted, interactive language in order to simplify access to its object oriented library written in C++.

In this work, we describe our experiences using Python,[5] a well established interpreted programming language, in parallel environments. We also present a set of packages that can be used to develop parallel FEM applications under Python.

The next section presents a brief overview of Python and related tools, and our Python ports to MPI, PETSc, ParMETIS, and OOFELIE. Section 3 describes design, implementation and provided functionality of these packages. Section 4 presents some efficiency comparisons between MPI for Python and C codes communicating numeric arrays. Section 5 shows a example of a parallel FEM simulation. Finally, section 6 presents our conclusions and plans for future work.

2 BACKGROUND

2.1 Python

Python[5,6] is a modern but mature, easy to learn, powerful programming language with a constantly growing community of users. It has efficient high-level data structures and a simple

but effective approach to object-oriented programming with dynamic typing and dynamic binding. Python's elegant syntax, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

The Python interpreter and its extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed. It can be easily extended with new functions and data types implemented in C or C++[7] and is also suitable as an extension language for customizable applications that require a programmable interface.

Python is an ideal candidate for writing higher-level parts of large-scale scientific applications and driving simulations in parallel architectures[8–10] like clusters of PC's or SMP's. Python codes are quickly developed, easily maintained, and can achieve a high degree of integration with other libraries written in compiled languages.

2.2 MPI

MPI,[11,12] the *Message Passing Interface*, is a standardized and portable message-passing system designed to function on a wide variety of parallel computers. The standard defines the syntax and semantics of library routines (MPI is not a programming language extension) and allows users to write portable programs in the main scientific programming languages (Fortran, C, or C++).

Since its release, the MPI specification has become the leading standard for message-passing libraries in the world of parallel computers. Implementations are available from vendors of high-performance computers as a component of the system software, and also from well known open source projects like MPICH[13,14] and LAM.[15,16]

MPI follows an object oriented design. Among the different abstractions introduced, *communicators* play the most important role. Basically, communicators specify a communication domain between an ordered set of processes or *group*. This abstraction enables division of processes, avoids message conflicts between different modules, and permits extensibility by users.

2.3 PETSc

PETSc[17–19] is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It employs the MPI standard for all message-passing communication.

PETSc is intended for use in large-scale application projects, and several ongoing computational science projects are built around the PETSc libraries. With strict attention to component interoperability, PETSc facilitates the integration of independently developed application modules, which often most naturally employ different coding styles and data structures.

PETSc is easy to use for beginners. Moreover, its careful design allows advanced users to have detailed control over the solution process. PETSc includes an expanding suite of parallel linear and nonlinear equation solvers that are easily used in application codes written in C, C++, and Fortran. PETSc provides many of the mechanisms needed within parallel application codes, such as simple parallel matrix and vector assembly routines that allow the overlap of

communication and computation. In addition, PETSc includes growing support for distributed arrays.

2.4 ParMETIS

ParMETIS[20] extends the functionality provided by METIS[21] and includes routines based on a parallel multilevel k-way graph-partitioning algorithms that are especially suited for parallel computations and large-scale numerical simulations involving unstructured meshes.

In typical FEM/FVM computations, ParMETIS dramatically reduces the time spent in inter-process communication by computing mesh decompositions such that the number of interface nodes/elements is minimized. In particular, ParMETIS provides functionalities for graph and mesh partitioning, dual graph construction, graph repartitioning, partitioning refinement, and sparse matrix reordering.

2.5 OOFELIE

OOFELIE, *Object Oriented Finite Elements Led by an Interactive Executor*, is a multi-physics software toolkit written in C++ with focus in modeling electro-thermo-mechanical systems. Simulation capabilities include fluids, heat transfer and phase change, stress and deformation, acoustics, and electrostatics. OOFELIE provides containers and routines to construct a system of linear or nonlinear equations to be solved for the unknowns of the physical problem.

The standard way of using OOFELIE is through scripts. Users define their problems in plain text files which are parsed and executed by a built-in interpreter. This interpreter also provides a command-line interface for interactive usage. OOFELIE can also be available as a library for development of new applications and has interfaces for pre- and post-processing software like *SAMCEF Field* and *GID*.

3 IMPLEMENTATION

Developed Python modules consist of Python code defining all constants, class hierarchies and functions. These codes call simpler, lower-level functions from extension modules written in C/C++, which provide access to native C/C++ objects, constants and function in libraries.

C/C++ extension modules were developed with the help of SWIG[22] wrapper generator. This tool simplifies the wrapping of C/C++ libraries as it automate the generation of complete extension modules by parsing header files and providing flexible customization mechanisms to define object conversion and type checking from C/C++ to Python and vice-versa. After a compilation step, generated extension modules are ready for access in the Python side.

4 EFFICIENCY MEASUREMENTS

Some efficiency tests were run on the Beowulf class cluster *Geronimo*[23] at CIMEC. Hardware consisted of ten computing nodes with Intel P4 2.4GHz processors, 512KB cache size, 1024MB RAM DDR 333MHz and 3COM 3c509 (Vortex) Nic cards interconnected with an Encore ENH924-AUT+ 100Mbps Fast Ethernet switch. All Python modules were compiled with

Python 2.4.1 and *Numarray 1.3.3*.

The first test was a bi-directional blocking send and receive between pairs of processors. MPI implementation used was *MPICH 1.2.6*. Messages were numeric arrays (*NumArray* objects) of double precision (64 bits) floating-point values. A basic implementation of this test using MPI for Python (translation to C or C++ is straightforward) is shown below.

```
from mpi4py import MPI
import numarray as array

rank = MPI.COMM_WORLD.Get_rank()
sbuff = array.array(shape=2**20,
                    type= array.Float64)
wt = MPI.Wtime()
if MPI.even:
    MPI.COMM_WORLD.Send(buffer, rank+1)
    rbuff = MPI.COMM_WORLD.Recv(rank+1)
else:
    rbuff = MPI.COMM_WORLD.Recv(rank-1)
    MPI.COMM_WORLD.Send(buffer, rank-1)
wt = MPI.Wtime() - wt

timing = MPI.WORLD.Gather(wt, root=0)
```

Results are shown in figure 1. Maximum bandwidth in Python is about 85% of maximum bandwidth in C.

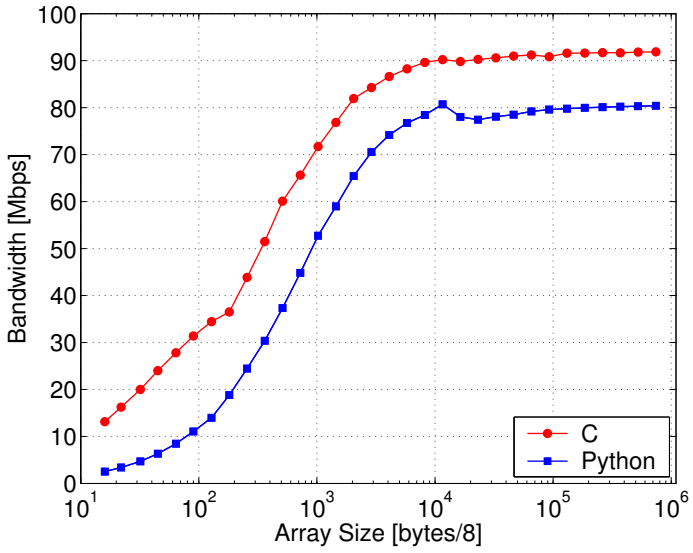


Figure 1: MPI for Python bandwidth in blocking Send/Receive

The second test was the sequential solution of a model transient three-dimensional heat transfer problem on the unit box $\Omega = (0, 1)^3$ with Dirichlet boundary conditions at the boundary Γ

and an uniform initial condition.

$$\begin{aligned} \dot{U} &= \nabla^2 U \text{ on } \Omega \times (0,T) \\ U(x,t) &= 0 \qquad \text{at } x \in \Gamma, 0 < t < T \\ U(x,t) &= 1 \qquad \text{at } x \in \Omega, t = 0 \end{aligned}$$

Finite differences with standard 7-points stencil on a structured, regularly spaced grid were used for the spatial discretization. Discrete Laplace operator was implemented with a matrix-free approach.

$$\begin{aligned} [L^h(U)]_{i,j,k} = & \frac{U_{i-1,j,k} - 2U_{i,j,k} + U_{i+1,j,k}}{h_1^2} + \\ & + \frac{U_{i,j-1,k} - 2U_{i,j,k} + U_{i,j+1,k}}{h_2^2} + \\ & + \frac{U_{i,j,k-1} - 2U_{i,j,k} + U_{i,j,k+1}}{h_3^2} \end{aligned}$$

Backward-Euler method for time integration was chosen. The solution process amounts to the solution of a system of linear equations in each time-step. Conjugate gradients algorithm without preconditioning is employed.

$$\frac{1}{\Delta t}\Big(U^{n+1} - U^n\Big) = L^h(U^{n+1}) \quad \rightarrow \quad \Big(\frac{1}{\Delta t} I - L^h\Big) U^{n+1} = \frac{1}{\Delta t} U^n$$

Results are shown in figure 2. Vertical axis indicates the Python overhead, horizontal axis indicates the number of grid points in each dimension. Clearly, the time overhead in using Python decreases when the problem size grows.

5 FEM SIMULATIONS

This section describes the numerical solution of a temperature-based model to simulate an unsteady heat conduction problem in a media undergoing mushy phase change.

During phase change, a considerable amount of latent heat is released or absorbed, causing a strong non-linearity in the enthalpy function. In order to model such phenomenon, we distinguish the different one-phase subregions at each side of the solidification front. Contributions from different phases are integrated separately in order to capture the sharp variations of material properties between phases. This approach, also called *discontinuous integration*, avoids regularizing the phenomenon and allows accurate evaluation of heat flux terms in discrete nonlinear equations.

The thermal model is first validated by solving a problem with analytical solution. Finally, results of a parallel simulation on a three-dimensional domain with linear tetrahedral finite elements are shown.

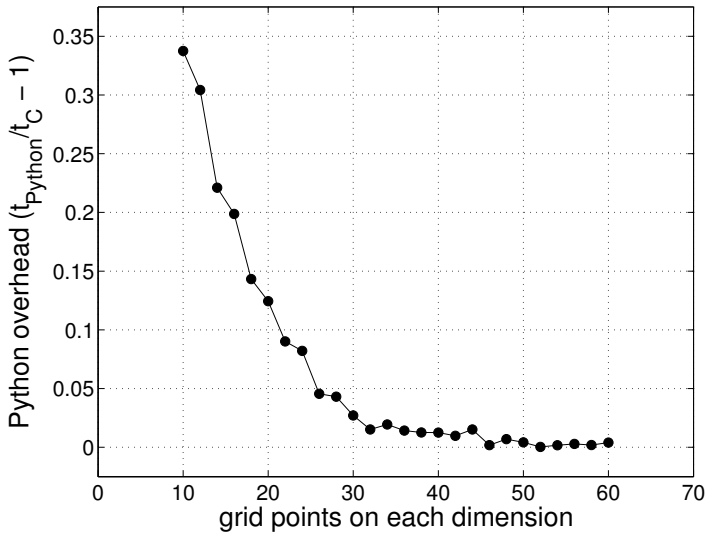


Figure 2: Efficiency measurements for a transient 3D heat transfer problem

5.1 Problem definition

Under assumptions of linear dependence of heat flux on temperature gradient (Fourier's law) and no melt flow during solidification process, the energy balance for each phase Ω_i is governed by equations of the form:

$$\rho\frac{\partial\mathcal{H}}{\partial t} - \nabla\cdot(\kappa\nabla T) = 0 \quad \forall(\boldsymbol{x},t)\in\Omega_i\times(0,\theta) \tag{1}$$

where T denotes the temperature, $\mathcal{H}$ the enthalpy (per unit volume) and $\kappa=\kappa(T)$ the material thermal conductivity (assumed isotropic). Equation (1) is supplemented by the following initial condition

$$T = T_0 \quad \forall\boldsymbol{x}\in\Omega_i, \quad t = t_0$$

and boundary conditions at $\partial\Omega$:

$$T = \bar{T} \qquad \text{at } \partial\Omega_T \tag{2}$$

$$-\kappa\nabla T\cdot\boldsymbol{n} = \bar{q} \qquad \text{at } \partial\Omega_q \tag{3}$$

$$-\kappa\nabla T\cdot\boldsymbol{n} = h_{env}(T - T_{env}) \qquad \text{at } \partial\Omega_c \tag{4}$$

being $\partial\Omega_T$, $\partial\Omega_q$ and $\partial\Omega_c$ non-overlapping portions of $\partial\Omega$, with prescribed temperature, conductive and convective heat flux, respectively. In the above, $\bar{T}$ and $\bar{q}$ refer to imposed temperature and heat flux fields, and T_{env} is the environment temperature, whose film coefficient is h_{env}; $\boldsymbol{n}$ denotes the unit outward normal to $\partial\Omega$.

Further, the following conditions must hold at the interface(s) Γ:

$$T = T_\Gamma \tag{5}$$

$$\langle\mathcal{H}u(\boldsymbol{\eta}) + \kappa\nabla T\cdot\boldsymbol{\eta}\rangle = 0 \tag{6}$$

where T_Γ is a constant value (for isothermal solidification, it is the melting temperature; otherwise, it is the solidus or liquidus temperature), $\langle * \rangle$ denotes the jump of the quantity $(*)$ in crossing the interface Γ, which is moving with speed u in the direction given by the unit vector $\boldsymbol{\eta}$. Note that the second equation states the jump energy balance at the interface.

In order to retrieve T as the only primal variable, we define the enthalpy as

$$\mathcal{H}(T) = \int_{T_{ref}}^{T} \rho c \mathrm{d}\tau + \rho \mathcal{L} f_l \tag{7}$$

being ρc and $\rho \mathcal{L}$ the unit volume heat capacity and latent heat, respectively, and T_{ref} an arbitrary reference temperature; f_l is a characteristic function of temperature, called volumetric liquid fraction, defined as

$$f_l(T) = \begin{cases} 0 & \text{if } T < T_{sol} \\ 0 \leq f_l^m(T) \leq 1 & \text{if } T_{sol} \leq T \leq T_{liq} \\ 1 & \text{if } T > T_{liq} \end{cases} \tag{8}$$

where T_{sol} and T_{liq} denote the solidus and liquidus temperatures, respectively, i.e., the lower and upper bounds of the mushy temperature range.

5.2 Numerical solution scheme

After obtaining the temperature-based form of of equation (1), spatial discretization is done using standard Galerking finite element method. This leads to a set of nonlinear system of ordinary differential equations, that is stated in matrix form as:

$$\boldsymbol{\Psi} = \boldsymbol{C}\frac{\partial \boldsymbol{T}}{\partial t} + \frac{\partial \boldsymbol{L}}{\partial t} + \boldsymbol{K}\boldsymbol{T} - \boldsymbol{F} = 0 \tag{9}$$

where $\boldsymbol{T}$ is the vector of unknown nodal temperatures, $\boldsymbol{C}$ the capacity matrix, $\boldsymbol{L}$ the latent heat vector, $\boldsymbol{K}$ the conductivity (stiffness) matrix and $\boldsymbol{F}$ the force vector.

Time integration in transient problems is done with the unconditionally stable first-order backward Euler method. This implicit scheme is applied on equation (9), which leads to a set of non-linear equations to be solved for the values of the temperatures at finite element nodes, at the end of the time increment considered:

$$\boldsymbol{\Psi}_{n+1} = \boldsymbol{C}_{n+1}\frac{\boldsymbol{T}_{n+1} - \boldsymbol{T}_n}{\Delta t} + \frac{\boldsymbol{L}_{n+1} - \boldsymbol{L}_n}{\Delta t} + \boldsymbol{K}_{n+1}\boldsymbol{T}_{n+1} - \boldsymbol{F}_{n+1} = \boldsymbol{0} \tag{10}$$

The solution of this highly non-linear discrete balance equation (10) is achieved by means of the well-known Newton-Raphson method. At each new iteration i, $\boldsymbol{\Psi}$ is approximated by using a first order Taylor expansion,

$$\boldsymbol{\Psi}_{(\boldsymbol{T}^{(i)})} \approx \boldsymbol{\Psi}_{(\boldsymbol{T}^{(i-1)})} + \boldsymbol{J}_{(\boldsymbol{T}^{(i-1)})}\Delta \boldsymbol{T}^{(i)} = \boldsymbol{0} \tag{11}$$

being $\boldsymbol{J} = \mathrm{d}\boldsymbol{\Psi}/\mathrm{d}\boldsymbol{T}$ the Jacobian or tangent matrix, and $\Delta\boldsymbol{T}^{(i)} = \boldsymbol{T}^{(i)} - \boldsymbol{T}^{(i-1)}$, the search direction. Iterative correction of temperatures is defined by:

$$\Delta\boldsymbol{T}^{(i)} = -[\boldsymbol{J}_{(\boldsymbol{T}^{(i-1)})}]^{-1}\boldsymbol{\Psi}_{(\boldsymbol{T}^{(i-1)})} \tag{12}$$

All terms in the tangent matrix for transient heat conduction can be found in classical texts,[24] and the latent heat contribution $\frac{\mathrm{d}\boldsymbol{L}}{\mathrm{d}\boldsymbol{T}}$ is detailed in Anca et al.[25]

5.3 A benchmark problem

Model validation has been performed by comparing analytical and numerical solutions for a transient non-linear heat transfer problem.[26]

This problem is concerned with solidification of a material which is initially at a temperature just above its freezing point and subject to a line heat sink in a infinite medium with cylindrical symmetry. The substance presents a wide freezing temperature range between the solidus and liquidus temperatures. Solid fraction is assumed to vary linearly with the temperature.

The cylindrical domain is initially at a uniform temperature T_0. The cylinder surface is maintained at a constant temperature which equals T_0. Because of symmetry, only a circular sector of the cylinder was discretized, forming a wedge.

As the material has a high latent heat, severe numerical discontinuities are present at the liquid-solid boundary. The use of a concentrated heat sink leads to large thermal gradients as r tends to zero. This singularity explains the error increment in the vicinity of the axis. Nevertheless, numerical results are in good agreement with analytical solution. In figure (3) we show exact and FEM adimensional temperature T/T_i together with the percentage error along the radius.

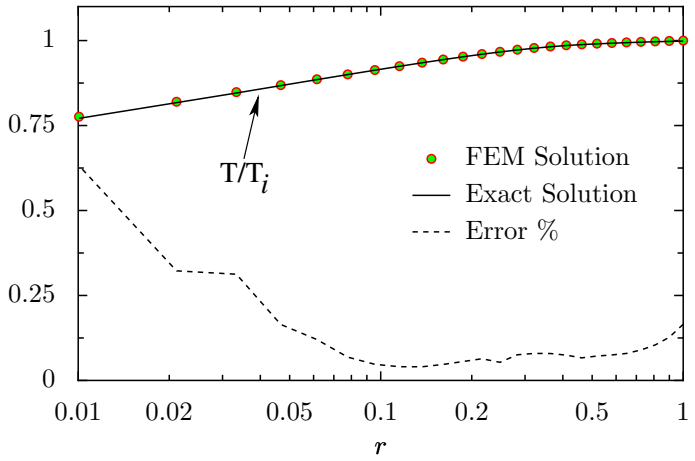


Figure 3: Analytical solution, FEM solution, and relative error of a model phase change problem

5.4 Parallel simulations

Using PETSc and OOFELIE modules, the solidification of an aluminum-copper alloy was simulated in parallel. This alloy has solidus and liquidus temperatures of approximately 540°C and 640°C respectively. Initial temperature was set to 800°C, temperature at the boundary was imposed to 200°C. The domain was a regular cube with 2 meters long edges. Because of symmetry, only one-eighth of the domain was discretized with a million degree of freedom. The mesh was obtained from a regular, structured mesh of hexahedra by splitting each hexahedron in six tetrahedra.

Results are shown in figure 4 for two representative time steps. Black lines are solidus and liquidus temperature isolines; they clearly indicate the separation of solid, mushy, and liquid phases.

6 CONCLUSIONS

Python is a very attractive language for rapid development of small scripts and code prototypes as well as large applications and highly portable and reusable modules and libraries. Unfortunately, like any scripting language, Python is not as efficient as compiled languages. However, it was conceived and carefully developed to be extensible in C (and consequently in C++). This exceptional characteristic enables Python to achieve performance in the time-critical parts of demanding applications. Moreover, Python can be used as a glue language capable of connecting existing software components in a high-level, interactive, and productive environment.

Efficiency tests have shown that performance degradation is not prohibitive, even for moderately sized problems. In fact, the overhead introduced is far smaller than the normal one associated to the use of interpreted versus compiled languages. Running Python on parallel computers is a good starting point for decreasing the large software costs of using HPC systems.

Future work will be directed towards the improvement of Python packages by extending they functionalities. Furthermore, the higher-level portions of our parallel multi-physics finite elements code PETSc-FEM[27,28] developed at CIMEC are planned to be implemented in Python in its next major rewrite. This work has already started and preliminary results are promising.

ACKNOWLEDGMENTS

This work received financial support from *Consejo Nacional de Investigaciones Científicas y Técnicas* (CONICET, Argentina), *Agencia Nacional de Promoción Científica y Tecnológica* (ANPCyT) and *Universidad Nacional del Litoral* (UNL) through grants CONICET-PIP-198 *Germen-CFD*, ANPCyT-PID-99/74 *FLAGS*, ANPCyT-FONCyT-PICT-6973 *PROA* and CAI+D-UNL-PIP-02552-2000.

The authors make extensive use of freely available software such as GNU/Linux operating system, GCC compilers, Python, Perl, MPICH and LAM/MPI implementations, PETSc libraries, METIS/ParMETIS libraries, Octave, Scilab, OpenDX and others. Many thanks to open source community for those excellent products.

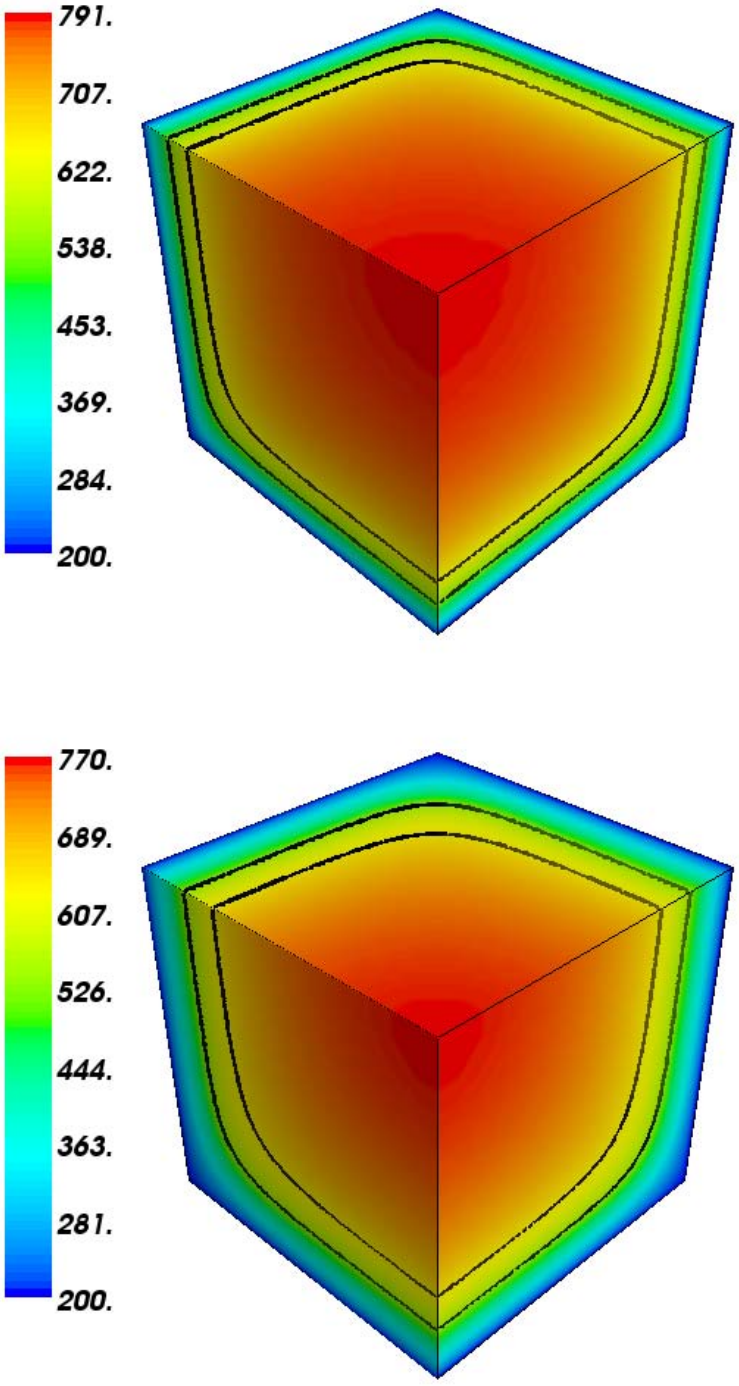


Figure 4: FEM solution: temperature (°C) for a phase change problem on a 3D domain

A OTHER EXAMPLES

A.1 Conjugate Gradients Algorithm

Figure 5 shows a snippet of code using PETSc to implement a basic version of conjugate gradients algorithm for the solution linear equation systems .

$$
\begin{aligned}
& i \Leftarrow 0 \\
& r \Leftarrow b - Ax \\
& d \Leftarrow r \\
& \delta_0 \Leftarrow r^T r \\
& \delta_{new} \Leftarrow \delta_0 \\
& \text{While } i < i_{max} \text{ and } \delta_{new} > \delta_0 \epsilon^2 \text{ do} \\
& \qquad q \Leftarrow Ad \\
& \qquad \alpha \Leftarrow \frac{\delta_{new}}{d^T q} \\
& \qquad x \Leftarrow x + \alpha d \\
& \qquad r \Leftarrow r - \alpha q \\
& \qquad \delta_{old} \Leftarrow \delta_{new} \\
& \qquad \delta_{new} \Leftarrow r^T r \\
& \qquad \beta \Leftarrow \frac{\delta_{new}}{\delta_{old}} \\
& \qquad d \Leftarrow r + \beta d \\
& \qquad i \Leftarrow i + 1
\end{aligned}
$$

```
def cg(A,b,x,imax=50,eps=1e-6):
    """
    A, b, x   : matrix, rhs, solution
    imax, eps : max iters, tolerance
    """
    r = b.Duplicate()
    d = b.Duplicate()
    q = b.Duplicate()
    i=0
    A.Mult(x,r); r.AYPX(-1,b)
    d.Copy(r)
    delta_0   = r.Norm()
    delta_new = delta_0
    while i<imax and \
              delta_new>delta_0*eps**2:
        A.Mult(d,q)
        alfa = delta_new/d.Dot(q)
        x.AXPY(alpha,d)
        r.AXPY(alpha,q)
        delta_old = delta_new
        delta_new = r.Norm()
        beta = delta_new/delta_old
        d.AYPX(beta,r)
        i= i+1
```

Figure 5: Basic conjugate gradients algorithm

A.2 One-dimensional Poisson's Problem

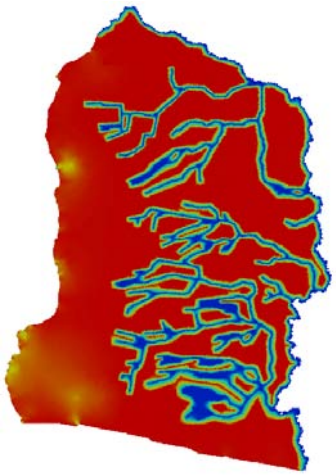
Figure 6 shows a Python script using PETSc for the numerical solution of 1D Poisson problem in the unit segment. Finite differences method is used for spatial discretization, discrete Laplace operator is implemented following a matrix-free approach, and the resulting linear system of equations is solved with conjugate gradients algorithm and Jacobi preconditioner.

A.3 Parallel Mesh Partitioning

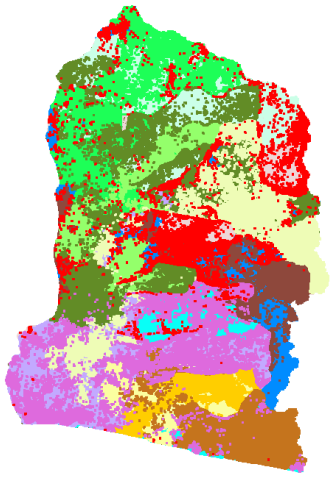
Figure 7 shows partitioning results obtained with ParMETIS for a unstructured mesh of 1.65 millions of triangular elements.

```
from petsc import PETSc

class Poisson1D:

    def __init__(self, N):
        self.h   = 1.0/(N+1)
        self.u_i = PETSc.ScalarArray(N)
        self.uxx = PETSc.ScalarArray(N)

    def Mult(self, U, U_xx):
        u_i, uxx = self.u_i, self.uxx
        U.GetArray(u_i)
        uxx[:]    = 0
        uxx[1:]  -= u_i[:-1]
        uxx[:-1] -= u_i[1:]
        uxx += 2 * u_i
        uxx /= self.h**2
        U_xx.SetArray(self.uxx)

    MultTranspose = Mult

    def GetDiagonal(self, D):
        D.Set(2/self.h**2)

nnods = 50
poisson1d = Poisson1D(nnods)
A = PETSc.Mat.CreateShell(nnods)
PETSc.MatShell.SetContext(A, poisson1d)
x, b = A.GetVecs()
b.Set(1)

ksp = PETSc.KSP('cg', pc_type='jacobi')
PETSc.Options.Set('ksp_monitor')
PETSc.Options.Set('ksp_vecmonitor')
ksp.SetFromOptions()
ksp.SetOperators(A)
ksp.Solve(b, x)
```

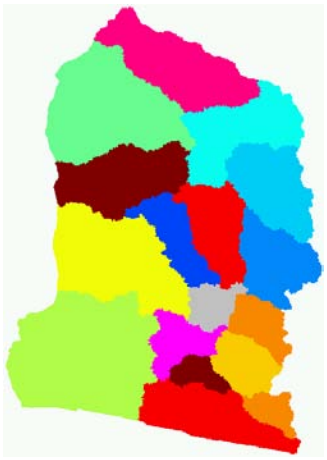
Figure 6: Matrix-free FDM for solving 1D Poisson equation



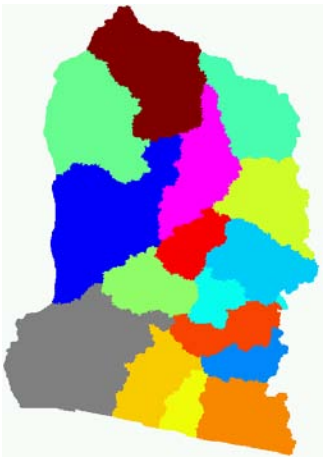
(a) mesh sizes, red=1000m, blue=50m



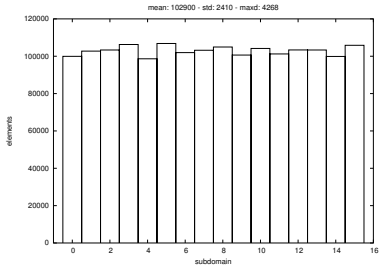
(b) initial mesh distribution



(c) sequential partition



(d) parallel partition



(e) partition quality

Figure 7: Partitioning of a mesh of triangular elements

REFERENCES

[1] Beowulf.org. The Beowulf cluster site, (2004). http://www.beowulf.org/.

[2] Message Passing Interface Forum. Message Passing Interface (MPI) Forum Home Page, (1994). http://www.mpi-forum.org/.

[3] Open Engineering. OOFELIE toolkit, (2004). http://www.open-engineering.com/.

[4] Alberto Cardona, Igor Klapka, and Michel Gerardin. Design of a new finite element programming environment. *Engineering Computation*, **11**(4), 365–381 (August 1994).

[5] Guido van Rossum. Python programming language, (1990–2004). http://www.python.org/.

[6] Guido van Rossum. Python documentation. http://docs.python.org/index.html, (May 2004).

[7] Guido van Rossum. Extending and embedding the Python interpreter. http://docs.python.org/ext/ext.html, (May 2004).

[8] SPaSM Team. SPaSM: Parallel molecular dynamics code, (1994–2001). http://bifrost.lanl.gov/MD/MD.html.

[9] David M. Beazley and Peter S. Lomdahl. Feeding a large scale physics application to Python. In *Proceedings of 6th. International Python Conference*, pages 21–29, San Jose, California, (October 1997).

[10] Konrad Hinsen. The Molecular Modelling Toolkit: A new approach to molecular simulations. *Journal of Computational Chemistry*, **21**(2), 79–85 (January 2000).

[11] MPI Forum. MPI: A message passing interface standard. *International Journal of Supercomputer Applications*, **8**(3/4), 159–416 (1994).

[12] MPI Forum. MPI2: A message passing interface standard. *High Performance Computing Applications*, **12**(1–2), 1–299 (1998).

[13] MPICH Team. MPICH: A portable implementation of MPI, (2004). http://www-unix.mcs.anl.gov/mpi/mpich/.

[14] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, **22**(6), 789–828 (September 1996).

[15] LAM Team. LAM/MPI parallel computing, (2004). http://www.lam-mpi.org/.

[16] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, (1994). http://www.lam-mpi.org/download/files/lam-papers.tar.gz.

[17] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc: Portable, extensible toolkit for scientific computation, (2001). http://www.mcs.anl.gov/petsc.

[18] Satish Balay, Victor Eijkhout, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools*

in Scientific Computing, pages 163–202. Birkhäuser Press, (1997).

[19] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, (2004).

[20] George Karypis. ParMETIS: Parallel graph partitioning and sparse matrix ordering, (1996–2005). `http://www-users.cs.umn.edu/~karypis/metis/parmetis/`.

[21] George Karypis. METIS: Family of multilevel partitioning algorithms, (1996–2005). `http://www-users.cs.umn.edu/~karypis/metis`.

[22] David M. Beazley. SWIG: Simplified wrapper and interface generator, (1996–2004). `http://www.swig.org/`.

[23] Mario A. Storti. Geronimo cluster at CIMEC, (2001–2004). `http://www.cimec.org.ar/geronimo`.

[24] Zienkiewicz O. and Taylor R. *The Finite Element Method*, volume 1: The Basis. Butterworth-Heinemann, 5th. edition, (2000).

[25] Andrés A. Anca, Alberto Cardona, and José M. Risso. 3d-thermo-mechanical simulation of welding processes. *Mecánica Computacional*, **XXIII**, 2301–2318 (2004).

[26] Özisik M. and Uzzell J. Exact solution for freezing in cylindrical symmetry with extended freezing temperature range. *Journal of Heat Transfer*, **101**, 331–334 (May 1979).

[27] Mario A. Storti, Norberto M. Nigro, and Rodrigo R. Paz. PETSc-FEM: A general purpose, parallel, multi-physics FEM program, (1999-2005). `http://www.cimec.org.ar/petscfem`.

[28] Victorio E. Sonzogni, Andrea M. Yommi, Norberto M. Nigro, and Mario A. Storti. A parallel finite element program on a Beowulf cluster. *Advances in Engineering Software*, **33**(7–10), 427–443 (October 2002).