On the Design of the EFCOSS Software Architecture When Using Parallel and Distributed Computing

Ralf Seidler, H. Martin Bücker, M. Ali Rostami and David Neuhäuser
Chair for Computer Architecture and Advanced Computing, Friedrich Schiller University, Jena, Germany

Keywords: Parallel and Distributed Computing, Simulation Software, Optimization Software, Software Frameworks, EFCOSS, Python, Fortran.

Abstract: Mathematical optimization algorithms are ubiquitous in computational science and engineering where the objective function of the optimization problem involves a complicated computer model predicting relevant phenomena of a scientific or engineering system of interest. Therefore, in this area of mathematical software, it is indispensable to combine software for optimization with software for simulation, typically developed independently of each other by members of separate scientific communities. From a software engineering point of view, the situation becomes even more challenging when the simulation software is developed using a parallel programming paradigm without taking into consideration that it will be executed within an optimization context. The EFCOSS environment alleviates some of the problems by serving as an interfacing layer between optimization software and simulation software. In this paper, we show the software design of those parts of EFCOSS that are relevant to the integration of a simulation software involving different parallel programming paradigms. The parallel programming paradigms supported by EFCOSS include MPI for distributed memory and OpenMP for shared memory. In addition, the simulation software can be executed on a remote parallel computer.

1 INTRODUCTION

Industry, science, and society are increasingly trying to model real-world problems using computer simulations. These computer models help to better understand, analyze, and predict complex phenomena arising from diverse application areas. Today, there is a strong and noticeable trend that carefully developed computer models are not only used to carry out a mere simulation of scientific and engineering systems, but they also serve—more and more—as the starting point for further investigations. For instance, scientists, engineers, and practitioners are interested in finding suitable values for input parameters of the computer model that are a priori unknown or that are only given with some level of uncertainty. Another important issue is to design a desired scientific or engineering system in a systematic way, i.e., by a goal-oriented design rather than by trial and error.

These investigations cannot start before the computer simulation of the given scientific or engineering system at hand is completed. That is, after having developed a sophisticated simulation software, whose predictions are thoroughly tested and validated against reality, the scientist, engineer, or practitioner is capable of going beyond a mere simulation of the reality. Finding input parameters and designing a scientific or engineering system are only two prominent examples of mathematical optimization problems. Indeed, in practical applications, there is an urgent need for the solutions of such optimization problems. Numerical techniques for the solution of optimization problems are available in a rich set of optimization software packages. Each of the underlying optimization algorithms has strengths and weaknesses (Nocedal and Wright, 2006; Dennis and Schnabel, 1983; Fletcher, 1987; Gill et al., 1981).

In a typical application scenario, there is a simulation software and an optimization software. The simulation software is typically developed by the community that is interested in some application area, for instance, computational fluid dynamics, computational electrodynamics, or bioinformatics. The optimization software, on the other hand, is usually developed by experts from numerical analysis or scientific computing. From a software engineering point of view, the challenge is now to bring together the software packages from these two different communities.

Seidler R., Bücker H., Rostami M. and Neuhäuser D..
On the Design of the EFCOSS Software Architecture When Using Parallel and Distributed Computing.
DOI: 10.5220/0004995804450454
In *Proceedings of the 9th International Conference on Software Engineering and Applications* (ICSOFT-EA-2014), pages 445-454
ISBN: 978-989-758-036-9

There are two common scenarios for interfacing these packages. The first scenario descents from the point of view of the simulation software. Here, an engineer working with a single simulation package is interested in using different optimization algorithms implemented in individual optimization software packages. The second scenario originates from the point of view of the optimization software. Here, a mathematician working with a single optimization package is interested in optimizing different engineering systems implemented in individual simulation packages. To combine simulation and optimization software, researchers have previously introduced a software framework called EFCOSS (Rasch and Bücker, 2010).

A related, but different approach is followed by the Toolkit for Advanced Optimization (TAO) (Munson et al., 2012; Benson et al., 2001; Kenny et al., 2004). This component-based optimization software is designed for the solution of large-scale optimization problems. TAO is capable of solving problems in the areas of nonlinear least squares, unconstrained minimization, bound constrained optimization, and general nonlinear optimization. It is not specifically designed for the solution of optimal experimental design (OED) problems. The software package VPLAN (Körkel, 2002) supports OED and parameter estimation for systems of differential algebraic equations. This software is mainly used for the solution of problems arising from process engineering. More related work is given in (Rasch and Bücker, 2010).

The new contribution of the present paper is twofold. First, we propose a novel software architecture for EFCOSS that is based entirely on Python. Second, we introduce to EFCOSS different ways of parallelism. Since, in real-world applications, the main computational effort is typically spent in the simulation software rather than in the optimization software, the focus of the present paper is on parallelism in the simulation software. In addition, parallelism can also be exploited in EFCOSS when solving multiple different optimization problems simultaneously (Seidler et al., 2014).

The structure of this paper is as follows. In Sect. 2, the new Python framework is introduced and an example demonstrating its use is shown in Sect. 3. Section 4 shows how EFCOSS interfaces with any simulation software that uses a parallel programming paradigm for distributed or shared memory. Section 5 then demonstrates how distributed computing is enabled where the simulation software is executed on a remote computer. Finally, Sect. 6 summarizes the findings of this paper and gives concluding remarks.

2 THE EFCOSS FRAMEWORK

The Environment for Combining Optimization and Simulation Software (EFCOSS) (Rasch and Bücker, 2010) is a software framework facilitating the solution of different types of optimization problems. Throughout this paper we consider an optimization problem of the form

$$\min_x \; g(x) \quad \text{subject to} \quad u(x) = 0, \tag{1}$$

where the symbols g and u denote smooth, real-valued functions on a subset of $\mathbb{R}^n$. Here, g is the objective function, while u is the equality constraint. For the sake of simplicity, we do not describe multiple equality constraints nor do we consider inequality constraints; both can also be handled by EFCOSS.

An illustrating example of an important class of optimization problems of type (1) consists of the data fitting problem described by the objective function

$$g(x) = \left\| D\big(d - f(x)\big) \right\|, \tag{2}$$

where

$$D := \operatorname{diag}(\omega_1, \omega_2, \ldots, \omega_m) \in \mathbb{R}^{m \times m} \tag{3}$$

is a diagonal matrix used to scale the entries of the residual vector

$$r(x) := D\big(d - f(x)\big) \in \mathbb{R}^m. \tag{4}$$

Here,

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

is a vector-valued function representing the simulation of a scientific or engineering problem of interest and the vector $d \in \mathbb{R}^m$ denotes some measurement data obtained for some property predicted by the function f.

We take this data fitting problem as a simple example to illustrate the functionality of EFCOSS. The overall structure of EFCOSS is depicted in Fig. 1. This high-level perspective is tailored toward the data fitting example. During the execution of an optimization algorithm, the optimizer requests the evaluation of the objective function g at a point x_0 from

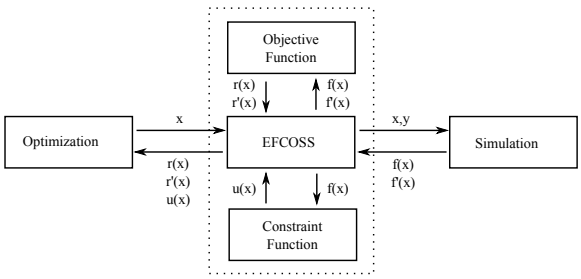

Figure 1: High-level EFCOSS architecture.

EFCOSS. Since the objective function needs the evaluation of the simulation f at the same point x_0, EFCOSS sends a corresponding request to the simulation software, possibly also transferring some additional data y that are necessary to run the simulation software. EFCOSS is also supporting the technology of automatic differentiation (AD) to compute derivatives of computer programs (Griewank and Walther, 2008; Rall, 1981). Given a computer program, an AD software tool automatically generates a new code capable of computing the derivatives of the function implemented by the given code. EFCOSS automatically generates interfaces for this AD code. So, EFCOSS gets not only the value of $f(x_0)$, but also its Jacobian matrix $f'(x_0) := \partial f/\partial x$ evaluated at the same point x_0. A similar procedure holds for the evaluation of the constraint function $u(x_0)$ and its derivative.

In addition to solving problems with an objective function of type (2), EFCOSS is also designed to solve more advanced optimization problems. In particular, it supports objective functions common in optimal experimental design (Pukelsheim, 2006), a topic which is not considered in the present paper.

EFCOSS initially relied entirely on distributed computing using the CORBA framework (Object Management Group, 2012). In practical applications solving real-world optimization problems, CORBA turned out to be particularly unpleasant for practitioners from outside of computer science. Since CORBA is also outdated and has several severe drawbacks (Henning, 2008), we removed CORBA. The new software design is based entirely on the flexible Python programming language using Numpy and Scipy data types and their primitives (Oliphant, 2007). Also, in the new design, the distributed approach is no longer a necessity, but can optionally be reinstalled by the use of Python Remote Object (PyRO) (de Jong, 2013); see the discussion in Sect. 5.

The implementation of EFCOSS consists of several Python classes. The main class of the framework is `EFCOSS`. In addition, the class `Simulation` serves as the Python interface to the simulation software. There are different interfaces for various optimization software packages. In the current version of EFCOSS, we provide interface codes for the following optimizers:

- ELSUNC (Wedin and Lindström, 1988; Lindström and Wedin, 1999),
- ENLSIP (Wedin and Lindström, 1988; Lindström and Wedin, 1999),
- FFSQP (Lawrence and Tits, 1996), and
- PORT (Gay, 1990).

There are also several freely available optimizers within the Scipy `optimize` package, which can also be used in a simple and straightforward way (The Scipy Community, 2013). We have successfully tested

- `fmin_cobyla`,
- `fmin_l_bfgs_b`, and
- `leastsq`.

In addition, there are some utility functions for generating interfaces to simulation codes and their derivatives.

3 USING EFCOSS

Let us examine EFCOSS by considering the following data fitting problem taken from the Minpack-2 test suite (Averick et al., 1992). Let

$$f_i(x) = x_1 + x_2 \cdot e^{-x_4 t_i} + x_3 \cdot e^{-x_5 t_i}$$

with $t_i = 10 \cdot (i-1)$ and $i \in \{1, \ldots, m\}$ denote some exponential functions. From these m scalar-valued components f_i, we construct the vector-valued function

$$f : \mathbb{R}^5 \to \mathbb{R}^m$$

that takes x_1, x_2, x_3, x_4 and x_5 as input. In this paper, this simple function is used to mimic an actual simulation software which would be much more complex in real-world applications. The minimization problem consists of (1) with the objective function (2) where $n = 5$ parameters are fitted and d is a given m-dimensional vector. To find a solution, different optimization software packages can be used. Here, we use the `opt_elsunc` interface to the optimizer ELSUNC.

In Fig. 2, a Fortran code for evaluating $f(x)$ is given by a subroutine called `sim`. The result of $f(x)$ is returned in the variable `fvec`.

If the minimization problem is solved for the first time the corresponding derivative code needs to be generated by an AD software tool. In this example, the derivative code is transformed by the AD tool

```
subroutine sim(x1,x2,x3,x4,x5,fvec,m)
  integer m
  double precision x1,x2,x3,x4,x5
  double precision fvec(m)
  integer i
  double precision temp,temp1,temp2

  do i = 1, m
       temp = dble(10*(i-1))
       temp1 = exp(-x4*temp)
       temp2 = exp(-x5*temp)
       fvec(i)=(x1+x2*temp1+x3*temp2)
  end do
end
```

Figure 2: A toy example of a simulation code taken from the Minpack-2 test collection.

```
SUBROUTINE SIM_DV(x1, x1d, x2, x2d, x3, x3d, &
& x4, x4d, x5, x5d, fvec, fvecd, m, nbdirs)
  USE DIFFSIZES
  IMPLICIT NONE
  INTEGER :: m
  DOUBLE PRECISION :: x1, x2, x3, x4, x5
  DOUBLE PRECISION, DIMENSION(nbdirsmax) :: x1d, &
& x2d, x3d, x4d, x5d
  DOUBLE PRECISION :: fvec(m)
  DOUBLE PRECISION :: fvecd(nbdirsmax, m)
  INTEGER :: i
  DOUBLE PRECISION :: temp, temp1, temp2
  DOUBLE PRECISION, DIMENSION(nbdirsmax) :: &
& temp1d, temp2d
  INTRINSIC DBLE
  INTRINSIC EXP
  DOUBLE PRECISION :: arg1
  DOUBLE PRECISION, DIMENSION(nbdirsmax) :: arg1d
  INTEGER :: nd
  INTEGER :: nbdirs
  DO nd=1,nbdirs
    fvecd(nd, :) = 0.D0
  END DO
  DO i=1,m
    temp = DBLE(10*(i-1))
    arg1 = -(x4*temp)
    DO nd=1,nbdirs
      arg1d(nd) = -(temp*x4d(nd))
      temp1d(nd) = arg1d(nd)*EXP(arg1)
      arg1d(nd) = -(temp*x5d(nd))
    END DO
    temp1 = EXP(arg1)
    arg1 = -(x5*temp)
    temp2 = EXP(arg1)
    DO nd=1,nbdirs
      temp2d(nd) = arg1d(nd)*EXP(arg1)
      fvecd(nd,i) = x1d(nd) + x2d(nd)*temp1 + x2*&
&       temp1d(nd) + x3d(nd)*temp2 + x3*temp2d(nd)
    END DO
    fvec(i) = x1 + x2*temp1 + x3*temp2
  END DO
END SUBROUTINE SIM_DV
```

Figure 3: Code automatically generated via the AD tool Tapenade from the code given in Fig. 2.

Tapenade (Hascoët and Pascual, 2013). The code `sim_dv` resulting from transforming the code in Fig. 2 is listed in Fig. 3.

An EFCOSS problem definition consists of a Python class used to steer all desired operations. This definition must be written by the user and, in our example, consists of the following methods:

- `initEFCOSS()` to initialize EFCOSS,
- `initAD()` to generate simulation interfaces,
- `initSim()` to initialize the simulation,
- `run_elsunc()` to execute the optimization algorithm.

In a runtime script, these methods can then easily be called as desired by the user. In addition, there is the possibility to interactively steer the execution from a console. Another option would be to write a graphical user interface on top of EFCOSS.

We now describe each of these methods in more detail. To initialize EFCOSS, we specify the variables and their values within EFCOSS. As shown in Fig. 4, this can be done with the method `initEFCOSS()` from the class `OptimizeSim`. Here, we first declare an instance of `EFCOSS` and assign the name `sim` for logging

```
from efcoss_utilities import *
from numpy import *
from EFCOSS import EFCOSS
class OptimizeSim:
  def initEFCOSS(self):
    self.opt=EFCOSS("sim")
    getEFCOSSRef(self.opt)
    m  = self.opt.newInputVariable("m",33)
    x1 = self.opt.newInputVariable("x1",0.5)
    # same for x2,x3,x4,x5
    ...
    fvec = self.opt.newOutputVariable("fvec",m)
    setSimulationCallingSequence([x1,x2,x3,x4,x5,
        fvec,m])
    setOptVars([x1,x2,x3,x4,x5])
    obj = self.opt.setObjectiveFunction("DataFit",
      "DataFit1d")
    indices,data,weights = readDataFile1d("data.
        txt")
    obj.addData1d(fvec,indices,data,weights)
```

Figure 4: Method `initEFCOSS()` to initialize EFCOSS.

purposes. Then, we set a global reference to that object using the method `getEFCOSSRef()`, followed by the definition of the input and output variables of the sample simulation `sim`, which evaluates the m components of the function $f(x)$. The dimension of the output variable `fvec` is defined by the input variable `m`. The input variables corresponding to the parameter vector x are defined using scalar floating-point values in this example. Then, the calling sequence of the simulation software is set according to the Fortran code of `sim` as given Fig. 2.

Next, the free parameters of the optimization need to be defined with the method `setOptVars()`. They are needed to generate the derivative code of `sim` via an AD tool. The last three statements are used to define the residual $r(x)$. Here, the one-dimensional data fitting problem represented by (2)–(4) is defined by the class `DataFit.DataFit1d` loaded by the method `setObjectiveFunction()`. The values of d are read by `readDataFile1d()` from the file `data.txt`. The vector d involved in the residual $r(x)$ is set by `addData1d()`, where the weights ω_i in (3) are all set to 1.0, since scaling is not necessary in this example. However, EFCOSS supports this form of scaling because, for more complicated problems, scaling can become crucial for numerical stability.

Next we discuss the method `initAD()`, also from the class `OptimizeSim`, which is given by

```
def initAD(self):
  from efcoss_codegen import generateServerIF
  AD_Tool="tapenade"
  generateServerIF("sim","simIF.f90",AD_Tool)
```

It generates the interfaces for the function `sim` and its derivative `sim_dv`. More precisely, the method `generateServerIF()` from the package `efcoss_codegen` generates this Fortran interface `simIF.f90`. This interface connects the simulation software written in Fortran to the EFCOSS class `Simulation` and consists of the subroutines `Func(iInput,dInput,Output)` and

`Jacobian(iInput,dInput,s_info,Output,Jac)`. The input vectors of the simulation and its derivatives are split into integer and double precision vectors denoted by `iInput` and `dInput`, respectively. Inside the interface, the values of these vectors are used to create input variables to the simulation. In the following text, we will merge these two input vectors into a single variable `Input` for better readability.

The results of the `Func` interface are copied to the vector `Output` in a linear fashion. The `Jacobian` interface returns the output in `dJac` and contains the additional input `s_info`. This array is an automatically generated vector necessary to generate the so-called seed matrix commonly used in AD. Given a seed matrix S, the AD-generated code computes the matrix-matrix product $f' \cdot S$ without explicitly computing the Jacobian f', thus potentially saving memory and storage. In this example, the seed matrix is set to the 5×5 identity matrix to compute f'.

The interface code is compiled and linked as follows:

1. `gfortran -c -fPIC sim.f90 sim_dv.f90`
2. `ar -r libsim.a sim.o sim_dv.o`
3. `f2py -c -m sim simIF.f90 -L. -lsim`

First, we compile the source `sim.f90` and its derivative `sim_dv.f90` with gfortran and put them into a static library `libsim.a`. This library can then easily be used with `f2py` when generating the shared object `sim.so` as the interface code to Python.

When the interface is compiled, EFCOSS needs to load this object. This is the purpose of the next method from the class `OptimizeSim`. This method `initSim()` reads as follows:

```
def initSim(self):
  from simulation import Simulation
  sim=Simulation(self.opt,"sim")
  self.opt.setSimulationServer(sim)
```

It gives the name of the object file `sim` to the `Simulation` class. This object is then made available to EFCOSS in the last statement.

From the class `OptimizeSim`, we finally discuss the method `run_elsunc()` depicted in Fig. 5. The ELSUNC optimizer can now be used to find an optimal solution of the data fitting problem. As a starting vector of the optimization, we use the initial values for $x_1, \ldots, x_5$ defined within EFCOSS. These can be retrieved by the method `getInitialValues()`. The length of the result vector of $g(x)$ is given by the function `lResidualVector()`. The remaining part of this method is used to steer the optimizer, e.g., the value of $p[2] = 100$ sets the maximal number of iterations to be carried out by the optimizer to 100.

Figure 6 illustrates the data flow from the optimization to the simulation software and back. The

```
def run_elsunc(self):
  import opt_elsunc
  elsunc=opt_elsunc.opt_elsunc(self.opt)
  initval = getInitialValues()
  n = len(initval)
  m = lResidualVector()
  x = array(initval,float64)
  mdc = m
  mdw = n*n+5*n+3*m+6
  p = array([0]*(11+2*n),int32)
  w = array([0.0]*mdw,float64)
  p[0] = 0
  p[1] = 6
  p[2] = 100
  p[4] = 2
  w[0:4] = 1.0E-6
  bnd = 1
  bl = array( [-100.0]*n, float64)
  bu = array( [ 100.0]*n, float64)
  info,fvec,cov = elsunc.elsunc(x,mdc,m,
      bnd,bl,bu,p,w)
```

Figure 5: Method `run_elsunc()` for executing the ELSUNC optimizer with the example.

optimizer for a data fitting problem needs to evaluate the residual vector r at the current parameter vector x. This is done by calling `evalfvec()`. The method `vectorfunction()` builds up the input vector `Input` to the simulation. This input vector consists of the values of x as well as additional input y that is defined by `initEFCOSS()`. In this example, the only additional input is the integer variable `m`. After calling the simulation via the Python interface `Function()`, the value of $f(x)$ is returned. The residual $r(x)$ is finally computed in `vectorfunction()`.

Again, we stress that, in real-world applications,

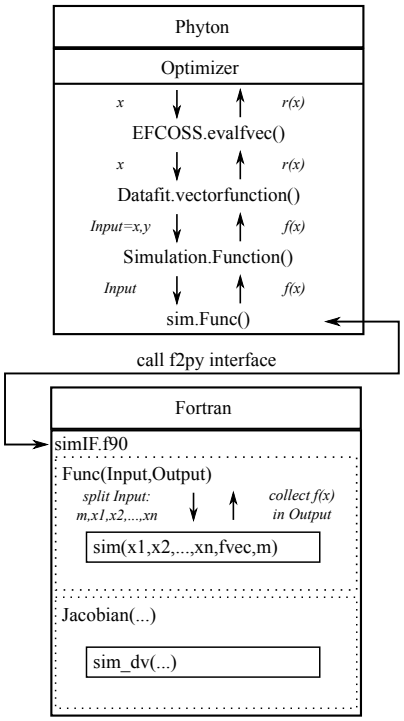

Figure 6: Schematic high-level overview of the data flow in an iteration of the optimization process.

the simulation code for evaluating the function $f(x)$ will be more complicated than the simplistic example sketched in Fig. 2. In an actual example from a geoscientific application which is described in more detail in (Seidler et al., 2014), the simulation code is given by

```
subroutine shemat(pP0,pK0,pN0,pTmax,pparm0,&
    pdhead,pdtemp,pdconc,pxcoord)
```

Here, the function $f(x)$ represents the computation of the head `pdhead`, temperature `pdtemp`, and concentration `pdconc` of a geothermal reservoir from a given set of geological parameters represented by `pparm0`. The code resulting from transforming this code via automatic differentiation reads

```
subroutine g_shemat_proc(ad_p_,pP0,pK0,pN0,pTmax,&
    pparm0,g_pparm0,pdhead,g_pdhead,pdtemp,&
    g_pdtemp,pdconc,g_pdconc,pxcoord,pZmin,&
    pZmax,omp_inner,omp_outer)
```

In this automatically generated code, the multidimensional arrays `g_pdhead`, `g_pdtemp`, and `g_pdconc` store the derivatives of head, temperature, and concentration with respect to the geological parameters. That is, these three arrays store the Jacobian matrix f'.

4 PARALLELIZED SIMULATION

Simulation software arising from real-world applications in science and engineering tend to require a large amount of computing time and storage. To cope with the long running times and the high storage requirements, it is often mandatory to run these simulations on parallel processors. For very large-scale problems, the simulation software has to be executed on a high-performance parallel computer typically installed at a computing center. For problems with a moderate storage requirement, parallelism is also relevant when the simulation is run on a single multicore workstation or laptop.

One of the strengths of EFCOSS is its tight integration with parallelized simulation software. That is, EFCOSS supports the solution of optimization problems where the simulation software is parallelized. More precisely, EFCOSS can integrate simulation software based on parallel programming paradigms for both shared-memory and distributed-memory systems. In this section, we will demonstrate how EFCOSS supports the two dominant parallel programming paradigms OpenMP and MPI.

OpenMP: Today, OpenMP is the de-facto standard for shared-memory parallel programming. This parallel programming paradigm is mainly used for a moderate number of concurrent threads. OpenMP consists

```
subroutine sim(x1,x2,x3,x4,x5,fvec,m)
...
!$OMP parallel
!$OMP do private(i,temp,temp1,temp2)
  do i = 1, m
       temp = dble(10*(i-1))
       temp1 = exp(-x4*temp)
       temp2 = exp(-x5*temp)
       fvec(i)=(x1+x2*temp1+x3*temp2)
  end do
!$OMP end do
!$OMP end parallel
end
```

Figure 7: OpenMP-parallelized version of `sim` from Fig. 2.

of a set of compiler directives and a runtime library. In OpenMP, parallelization is carried out by inserting certain directives to a serial code (Chapman et al., 2008; OpenMP Architecture Review Board, 2013). Therefore, an OpenMP-parallelized program differs only slightly from the serial program. Consider the code in Fig. 7 as an illustrative example. This figure shows the OpenMP-parallelized version of the simulation code given in Fig. 2. In this example, the `for` loop does not depend on the order of its iterations. Therefore, it can be parallelized. The iterations of the loop are distributed to a team of threads. In shared-memory parallel programming, each OpenMP thread has access to the result vector `fvec`. However, the variables specified in the `private` clause are replicated as local copies to each thread. Since each thread is writing to a different part of the vector `fvec` the parallelization is correct.

To solve a single optimization problem, EFCOSS is executed in a single process p. Therefore, calling a function from an OpenMP-compiled library is feasible. Since an OpenMP-parallelized program is finalized with a barrier at the end of a parallel region, the computation of the process p is resumed without synchronization problems. Notice that the OpenMP flag is needed in the compilation as well as in the f2py command building the interface. The complete command sequence is as follows:

1. `gfortran -c -fopenmp -fPIC sim.f90 sim_dv.f90`
2. `ar -r libsim.a sim.o sim_dv.o`
3. `f2py --f90flags='-fopenmp' -c -m sim simIF.f90 -L. -lsim -lgomp`

MPI: Another parallel programming model which is mostly employed in large-scale applications is the distributed-memory paradigm. Currently, the dominant distributed-memory programming paradigm is the Message Passing Interface (MPI) (Snir et al., 1995; Snir et al., 1998; Gropp et al., 1998). In comparison to OpenMP, programs written in MPI require major changes to the serial code. These changes are required, as the whole data structure of the program,

```
subroutine sim(x1,x2,x3,x4,x5,fvec,m)
use mpi
integer m,local_m,prank,psize
double precision x1,x2,x3,x4,x5,fvec(m)
double precision, allocatable:: local(:)
integer i
double precision temp, temp1, temp2
real t1,t2
call mpi_comm_rank(mpi_comm_world,prank,ierr)
call mpi_comm_size(mpi_comm_world,psize,ierr)
local_m = m/psize
allocate(local(local_m))
do i = 1, local_m
  k=i+prank*local_m
  temp = dble(10*(k-1))
  temp1 = exp(-x4*temp)
  temp2 = exp(-x5*temp)
  local(i)=(x1+x2*temp1+x3*temp2)
end do
call mpi_gather(local,local_m,&
  & MPI_DOUBLE_PRECISION,fvec,local_m,&
  & MPI_DOUBLE_PRECISION,MPI_IN_PLACE,&
  & MPI_COMM_WORLD,ierr)
end
```

Figure 8: MPI parallelization of the code given in Fig. 2.

specifically all arrays, have to be explicitly decomposed. Communication between different processes is specified via send and receive commands. Here, since an array is distributed over different processes, each process has access to its own part of that array. To access other parts of the array, communication is needed between processes. Hence, any operation which needs the whole array is a bottleneck. However, in our example, each process computes its own local part independently and the communication is only needed to gather all parts of the array at the end.

Figure 8 shows the MPI-parallelized code of the example given in Fig. 2. Here, the array `local` of size `local_m` is the working array of each process. The MPI function `MPI_GATHER` gathers the results of the separate processes and stores it in the array `fvec` of the master process. The derivative subroutine `sim_dv()` is parallelized in a similar way. However, in a real-world MPI code, the programmer might have used calls to the `mpi_init()` and `mpi_finalize()` routines in its simulation code. These calls need to be commented out beforehand to prevent errors. The Python package `mpi4py` takes care of these calls (Dalcín et al., 2005).

EFCOSS is capable of generating MPI-enabled interface code, by adding `mpi=1` to the `generateServerIF()` method. We also implemented another class, `SimulationMPI`, that adds MPI functionality to the Python simulation code. This class has to be used in the `initSim()` method instead of the class `Simulation`.

The idea of using MPI-parallelized simulations in EFCOSS is to use a master-worker principle. The execution of the optimization algorithm is started in just one MPI process whose MPI rank is 0. We refer to this process as the master process. The remaining MPI processes are used as worker processes that create a basic EFCOSS instance and execute the method `run_function_worker()`. The corresponding worker code is presented in Fig. 9. The master process is supposed to execute the method `run_elsunc()`.

```
def run_function_worker(self):
  x=getInitialValues()
  res=self.opt.evalfjac(0,x,33)
```

Figure 9: Running the simulation as a worker MPI-process.

Let us now describe the situation in the interfaces for the master and workers with the help of Fig. 10. The master process is executed as described before. It first evaluates the subroutine `Func()` and then the derivative subroutine `Jacobian()`. As described in Fig. 6, these two routines are used by the optimizer to compute the next parameter vector x. In addition, the master is responsible for sending a variable `k` that is used to control the different tasks executed by the workers. More precisely, the value `k=1` is used to signal that the workers will execute the simulation. The value `k=2` represents the execution of the derivative code. When the master is terminated, it sends to the workers the value `k=0` to signal the termination of their programs as well.

A worker, on the other hand, starts the `evalfjac()` method from the EFCOSS class. That is, it starts the Jacobian interface wrapper `Jacobian()`. In this interface, the worker at first receives the variable `k`. After receiving the value `k=1` the worker enters a loop that can only be exited when the master sends `k=0`. Next, the master's `input` vector is sent to all workers. Depending on the value of `k`, all the processes start either the simulation code `sim()` or its derivative code `sim_dv()`. In the case of the derivative code, the master also sends the vector `s_info` to the workers. The workers receive this additional input to the derivative code before they start the derivative code. Finally, the workers wait for the next `k` and the master continues with the optimization algorithm.

The user must take care of the correct starting of the master and workers. An example of a runtime script for this purpose is shown in Fig. 11. First, the MPI environment is set up by retrieving the `rank` of the process on which the MPI code is executed. Then, EFCOSS is initialized and the simulation is set up. The master process starts its optimization with ELSUNC, while the workers start the `run_function_worker()` method. When the optimization algorithm is finished, the `Finalize()` method of the `SimulationMPI` class is executed. So, the workers are also terminated.

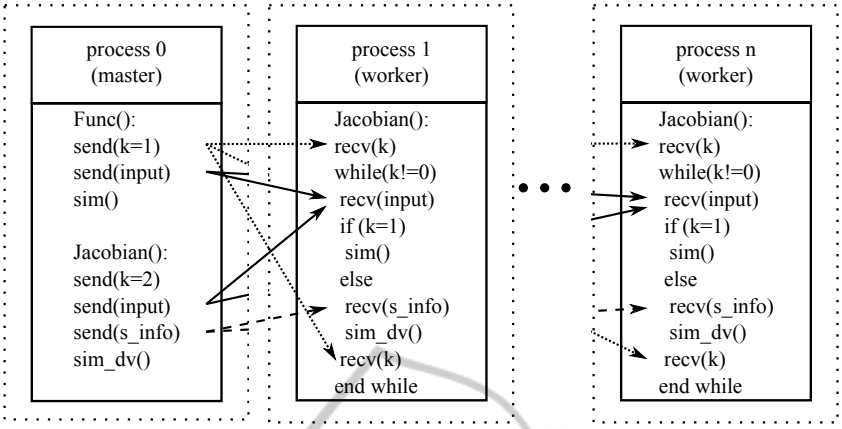

Figure 10: Using Fortran simulation interfaces in conjunction with MPI.

```
from OptimizeSim import OptimizeSim
from mpi4py import MPI

comm=MPI.COMM_WORLD rank=comm.Get_rank()

d=OptimizeSim() d.initEFCOSS() d.initSim()

if (rank==0):
   d.run_elsunc()
   d.opt.Simulation.Finalize()
else:
   d.run_function_worker()
```

Figure 11: Runtime script for the solution of an optimization problem where the simulation is parallelized with MPI.

5 REMOTE SIMULATION

When the focus is on large-scale problems, it is not uncommon that an optimization or simulation software is tuned for a particular high-performance computing system. In general, not all the optimization and simulation software packages are available on all systems. We therefore suggest an approach based on distributed computing. This enables the combined use of (i) a dedicated simulation workstation or cluster, (ii) a dedicated optimization workstation or cluster, and (iii) a dedicated steering workstation. The resulting remote object principle is shown in Fig. 12.

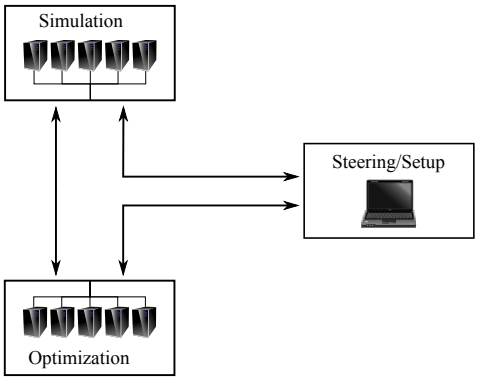

Figure 12: Remote object principle illustrating the general possibilities for distributed computing in EFCOSS. (Cliparts from openclipart.org.)

In this paper we focus on parallelism involved in the simulation. We could also envision a situation where the optimization algorithm is executed in parallel. However, the overall runtime is typically dominated by the simulation and its derivatives rather than by the optimization algorithm. So, we do not discuss in this paper a distributed approach for a parallel optimization algorithm. Rather we consider distributed computing for a parallel simulation. To this end, let the optimization algorithm as well as the steering and setup be serially executed on some machine. The simulation is then run on a different server, using an MPI-parallelized code as discussed in the previous section.

We use the version 4.24 of the Python Remote Object (PyRO) library (de Jong, 2013) for distributing Python objects. An example starting a simulation server is shown in Fig. 13. This example shows the MPI server code that starts a PyRO daemon on the master process (`rank=0`). The other processes just initialize the `OptimizeSim` class as stated before and start the runtime loop with `run_function_worker()`, shown in Fig. 9. As of PyRO version 4.18, the default serializer for sending objects is set to `serpent`. This serializer is currently not capable of transferring numpy data types. To overcome this issue, we continue to use the `pickle` serializer instead. Pickling is a standard serialization protocol in Python, which has drawbacks with respect to security. We assume that this issue is solved in the next versions of PyRO so that the `serpent` serializer can also transfer numpy data types.

To correctly communicate with the remote simulation server, the EFCOSS instance running the optimization algorithm needs a new initialization method

```
from simulation import SimulationMPI
from OptimizeSim import OptimizeSim
import Pyro4
from mpi4py import MPI
Pyro4.config.SERIALIZER='pickle'
Pyro4.config.SERIALIZERS_ACCEPTED.add(
    'pickle')

def main():
    comm=MPI.COMM_WORLD
    rank=comm.Get_rank()
    size=comm.Get_size()
    if (rank==0):
      simulation=SimulationMPI(modname="sim")
      Pyro4.Daemon.serveSimple(
        {simulation:
          "efcoss.Simulation"},
        ns=False)
    else:
       d=OptimizeSim()
       d.initEFCOSS()
       d.initSim()
       d.run_function_worker()

if __name__=="__main__":
    main()
```

Figure 13: Python code for a remote MPI simulation server using PyRO.

for the simulation. A corresponding code is given in Fig. 14. This is a simple example illustrating the creation of a server process. Here, PyRO is used without a nameserver. The PyRO object of the simulation is retrieved by entering its universal resource identifier (URI), which the daemon prints on the remote machine.

```
def initSimRemote(self):
  uri=input("Enter␣URI␣of␣Simulation␣Server:␣").
       strip()
  sim=Pyro4.Proxy(uri)
  sim.setResultVec(self.opt.getResultVec())
  sim.setJacVec(self.opt.getJacVec())
  self.opt.setSimulationServer(sim)
```

Figure 14: Python code to retrieve a remote object from the simulation server.

6 CONCLUSIONS

An important class of problems arising in science and engineering is to solve mathematical optimization problems including data fitting problems, where a suitably-defined objective function is minimized. A typical objective function involves the evaluation of a mathematical function that is represented by a complicated simulation code. The scenario we consider in this paper consists of bringing together an optimization software package with a simulation package via the simple and user-friendly software framework EFCOSS, the Environment for Combining Optimization and Simulation Software.

Today, many scientific and engineering software packages involve some sort of parallelism. The most prominent parallel programming paradigms are OpenMP for shared-memory computers and MPI for systems with distributed memory. With the help of an illustrative example, we presented the feasibility of using EFCOSS to solve optimization problems that involve simulation codes using either OpenMP or MPI. Though not explicitly described, there is room for another viable approach in which a combination of OpenMP and MPI is used in a hybrid fashion. Furthermore, EFCOSS facilitates distributed computing by means of the publicly available Python Remote Object package PyRO.

Similar to the coupling of a simulation software package to EFCOSS, it is also relevant to consider the situation where developers of optimization packages are interested in interfacing their software with a simulation using EFCOSS. This is an ongoing work which will be described elsewhere.

ACKNOWLEDGEMENTS

This work is partially supported by the German Federal Ministry for the Environment, Nature Conservation and Nuclear Safety (BMU) within the project MeProRisk II, contract number 0325389 (F) as well as by the German Federal Ministry of Education and Research (BMBF) within the project HYDRODAM, contract number 01DS13018.

REFERENCES

Averick, B. M., Carter, R. G., Moré, J. J., and Xue, G.-L. (June 1992). The MINPACK-2 test problem collection. Technical Report MCS–P153–0692, Argonne National Laboratory.

Benson, S. J., Curfman McInnes, L., and Moré, J. J. (2001). A case study in the performance and scalability of optimization algorithms. *ACM Transactions on Mathematical Software*, 27(3):361–376.

Chapman, B., Jost, G., Van der Pas, R., and Kuck, D. J. (2008). *Using OpenMP: Portable shared memory parallel programming*. MIT Press, Cambridge, Mass., London.

Dalcín, L., Paz, R., and Storti, M. (2005). MPI for Python. *Journal of Parallel and Distributed Computing*, 65(9):1108–1115.

de Jong, I. (2013). Pyro – Python Remote Objects. http://pythonhosted.org/Pyro4.

Dennis, Jr., J. E. and Schnabel, R. B. (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Englewood Cliffs.

Fletcher, R. (1987). *Practical Methods of Optimization*. John Wiley & Sons, New York, 2nd edition.

Gay, D. M. (1990). Usage summary for selected optimization routines. Computing Science Technical Report 153, AT&T Bell Laboratories.

Gill, P. E., Murray, W., and Wright, M. H. (1981). *Practical Optimization*. Academic Press, New York.

Griewank, A. and Walther, A. (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition.

Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E. L., Nitzberg, B., Saphir, W., and Snir, M. (1998). *MPI–The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT Press, Cambridge, MA, USA.

Hascoët, L. and Pascual, V. (2013). The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Softw.*, 39(3):20:1–20:43.

Henning, M. (2008). The rise and fall of CORBA. *Communications of the ACM*, 51(8):52–57.

Kenny, J. P., Benson, S. J., Alexeev, Y., Sarich, J., Janssen, C. L., Curfman McInnes, L., Krishnan, M., Nieplocha, J., Jurrus, E., Fahlstrom, C., and Windus, T. L. (2004). Component-based integration of chemistry and optimization software. *Journal of Computational Chemistry*, 25(14):1717–1725.

Körkel, S. (2002). *Numerische Methoden für Optimale Versuchsplanungsprobleme bei nichtlinearen DAE-Modellen*. PhD thesis, University of Heidelberg, Germany.

Lawrence, C. T. and Tits, A. L. (1996). Nonlinear equality constraints in feasible sequential quadratic programming. *Optimization Methods and Software*, 6:265–282.

Lindström, P. and Wedin, P.-Å. (1999). Gauss-Newton based algorithms for constrained nonlinear least squares problems. Department of Computing Science, Faculty of Science and Technology, Umeå University, Sweden.

Munson, T., Sarich, J., Wild, S., Benson, S., and Curfman McInnes, L. (2012). TAO 2.0 users manual. Technical Report ANL/MCS–TM–322, Mathematics and Computer Science Division, Argonne National Laboratory. http://www.mcs.anl.gov/tao.

Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization*. Springer, New York, 2nd edition.

Object Management Group (2012). Common Object Request Broker Architecture (CORBA): Specification, Version 3.3. http://www.omg.org/spec/CORBA/3.3.

Oliphant, T. E. (2007). Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20.

OpenMP Architecture Review Board (2013). OpenMP Application Program Interface, Version 4.0. http://www.openmp.org.

Pukelsheim, F. (2006). *Optimal Design of Experiments*. Number 50 in Classics in Applied Mathematics. SIAM, Philadelphia.

Rall, L. B. (1981). *Automatic Differentiation: Techniques and Applications*, volume 120. Springer Verlag, Berlin.

Rasch, A. and Bücker, H. M. (2010). EFCOSS: An interactive environment facilitating optimal experimental design. *ACM Transactions on Mathematical Software*, 37(2):13:1–13:37.

Seidler, R., Bücker, H. M., Padalkina, K., Herty, M., Niederau, J., Marquart, G., and Rasch, A. (2014). Redesigning the EFCOSS framework towards finding optimally located boreholes in geothermal engineering. In Horvát, I. and Rusák, Z., editors, *Proceedings of the tenth international symposium on tools and methods of competetive engineering (TMCE 2014), May 19–23, 2014, Budapest, Hungary*, pages 831–842.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. (1995). *MPI–The Complete Reference*. MIT Press, Cambridge, MA, USA.

Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W., and Dongarra, J. (1998). *MPI–The Complete Reference: Volume 1, The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd edition.

The Scipy Community (2013). SciPy v0.13.0 reference guide.

Wedin, P.-Å. and Lindström, P. (1988). Methods and software for nonlinear least squares problems. Technical Report UMINF–133.87, Inst. of Information Processing, University of Umeå, Umeå, Sweden.