Stanford Exploration Project, Report 120, May 3, 2005, pages 41–62

Stanford Exploration Project, Report 120, May 3, 2005, pages 41–62

Inversion and fault tolerant parallelization using Python

Robert G. Clapp[1]

ABSTRACT

Many current areas of research at SEP involve large-scale inversion problems that must be parallelized in order to be tractable. Writing fault-tolerant, parallel code requires significant programming expertise and overhead. In this paper, a library, written in Python, is described that effectively simulates a fault-tolerant parallel code, using simple serial programs. In addition, the library provides the ability to use these parallel objects in out-of-core inversion problems in a fault-tolerant manner.

INTRODUCTION

The large size of today's oil industry problems necessitates harnessing the power of clusters. The problem is that as we add nodes, we increase our odds of node failure. Inversion on large-scale problems is even more problematic. Operators can take days to weeks to run (Sava and Biondi, 2003; Clapp and Clapp, 2005) and can involve multiple instances of complex operations (Clapp, 2003). Running these problems on Beowulf clusters poses a problem as the odds of a multi-week job running without a node failing are low.

In Clapp (2004a), I described a library, written in Python, that allows auto-parallelization with a high-level of fault tolerance for almost any SEPlib program. Instead of handling parallelization within a compiled code at the library level, the parallelization is done at the script level which sits on top of the executables. The Python library distributes and collects the datasets, keeps track of what portions of the parallel job are done, and monitors the state of the nodes. The distribution and collection are done through MPI but individual jobs are all serial codes. The code is written using Python's object-oriented capabilities so it is easily expandable. A parallel job is described by a series of files and a series of tasks.

For inversion problems, Clapp (2004b) describes a Python inversion library which uses abstract vector and operator descriptions. From these abstract classes I derive specific classes to handle out-of-core problems. Operators become wrappers around SEPlib programs and vectors wrappers around SEPlib files.

In this paper I introduce an improved version of the library described in Clapp (2004a) and Clapp (2004b). The new version provides significant additional flexibility. Multiple programs can be combined into single executables. Parallel files can now be SEP3D files, and/or involve

[1] **email:** bob@sep.stanford.eda

overlapping patches. Inversion can be done on parallel files (instead of collected on some master node), saving disk space and transfer time.

In the first portion of the paper I will cover the basic Python parallel and inversion objects. In the second portion I will show several examples on how to use these objects to accomplish tasks that are both memory and computationally intensive.

BASIC BUILDING BLOCKS

Before delving into parallel and inversion objects, it is useful to go over some core objects that are used extensively by the more advanced parallel and inversion objects. These objects provide interaction with the command line, SEP files, and keep track of what the program has accomplished and what needs to be done.

Parameter

The `SEP.parameter.parameter` is a holding class for information about a parameter that can be described by an ASCII string. It can store documentation about the parameter, its current value, and/or some default value for the parameter. In addition to providing a simple mechanism for accessing arbitrary objects, it is also useful for documenting programs/objects and checking to make sure all parameters required for a given routine have been set.

Parameters

The `SEP.args.basic` object can be thought of as a collection of parameters accessible using SEP programming conventions. You can request a parameter using `par.par("tag")`, optionally with a default value, and an error if it doesn't exist. You can also add a parameter `par.add_param("tag",value)` to the collection. The `SEP.args.basic` can be initialized from a file, another collection of parameters, or an empty set can be created. It has the ability to output its contents in various manners. For example, it can return a simple dictionary linking parameters to their values or can return the parameters in a sep-style convention (par=val) in either a list or string form. It can also write its contents into a file.

Sepfile

A SEPlib file object `SEP.sepfile.sep_file` is built from one to three `SEP.args.basic` objects (history file, header format file, and grid format file). You can access the collection of parameters directly through the history, headers, and grid objects (e.g. `sep.history.par("n1")`). In addition it has the concept of axes and keys (if the data is SEP3D and possesses headers). You can get the number of dimensions, retrieve and set axes (n,o,d,label,unit) and keys (name,format,type), and return the size of the dataset.

Status

The status object `SEP.stat_sep.status` keeps track of the progress of a job. It is a dictionary linking a job descriptor to a list of properties of that job descriptor. For example, when doing a parallel job, the list will include the status of the job *todo, sent, running, finished, collected*, the machine the job is run on, the progress of the job,where the stderr and stdout of the job is stored, and how many times the job has failed to run correctly. The status information is written to an ASCII file that enables a job to be restarted automatically. The class also has the ability to store parameter information in the status file which can be automatically read at startup.

Options, Flows, and Programs

Programs generally follow a fairly standard flow. First you read in parameters from the command line, then process these parameters, potentially creating new parameters, and finally you run the algorithm. You can think of a flow as series of programs, where you run through these basic steps for each program. An example is wave-equation migration. To perform wave-equation migration a typical flow would be to choose the reference velocities, convert the data to the frequency domain, migrate the data, transpose the data, and create angle gathers. All of these steps could be independent programs, combined into a single program, or a multitude of other possible variations. The advantage of combining these steps into a single program is that many times they share parameters, so you are simplifying the user's jobs. The disadvantage is the code becomes more difficult to manage.

The options, flow, and program objects attempt to allow the simplicity of simple programs that perform a single operation, with the advantage of a shared parameter space. A `SEP.opt_base.options` is an extension of the `SEP.args.basic` class. It has several additional abilities:

- It can output self-documentation based on its set of parameters.
- It introduces five additional functions:
  - The `read_params(`*args,prefix*`)` extracts from the `SEP.args.basic` object all of the parameters that the options group has been initialized with. If the parameter is required and doesn't exist it returns an error with the self-doc for the parameter. The prefix argument will limit the search of *args* to parameters that begin with *prefix*, everything after the prefix will be assumed to be the name of the parameter for the object.
  - `build_check_params()` which builds additional parameters and checks parameter validity based on command line arguments.
  - The function `prep_run(restart)` runs the task associated with the parameter group. Examples might be running a serial code or a parallel job.

- Finally `clean_files()` is meant to remove any unneeded files after the execution of the job.

The `SEP.opt_base.options` is currently inherited by four classes. The `SEP.opt_none.options` is for a group of options that aren't associated with a job. The `prep_run` and `clean_files` routines are by default empty. The `SEP.opt_prog.options` is for a set of options associated with a serial code. The `prep_run` function executes the code with the parameters associated with the object. The final two children, `SEP.par_job.par_job` and `SEP.solv_base.solver`, are discussed later.

The `SEP.flow.flow` object is a collection of parameter groups and flows. In the migration example we might have the velocity selection and migration as independent programs, where each has a parameter group associated with them. The angle gathers might be a flow composed of transposing the data and then creating the angle gathers. The `SEP.opt_flow.flow` object is initialized with a set of flows, a set of parameter groups, the order in which to run them, and potentially prefixes associated with the individual flows and parameter groups. The *prefixes* argument is a dictionary linking a given parameter group or flow to the prefix that all of its parameters will be initialized with.

The flow object has two basic functions: `add_options` and `prep_run`. The first function takes in a set of parameters *args* and runs `read_params(`*args,prefix*`)`. It then runs `build_check_params` on each parameter group and flow.

The `prep_run` call in a flow `prep_run` and `clean_files` parameter groups and flows. It is also regulated by a `SEP.stat_sep.status` object that allows that job to be restarted.

The final object, `SEP.prog.prog` inherits from the flow object. In addition, it has the concept of description and usage blocks for documentation, and by default uses the command line arguments when parsing parameters.

PARALLEL OBJECTS

The library is currently designed for coarse-grain parallel jobs that fit on a single processor (no inter-process communication necessary). The user writes a serial code that works on a portion or all of the dataset. Each parallel job is broken up into a series of *tasks*. These *tasks* can have the same, or different parameters, and the various input and output files can be distributed in several different manners. There are three basic classes of parallel objects: core objects that handle communication, parallel file objects that describe how a given file is distributed and parallel job objects that handle distributing the various tasks.

Parallel building blocks

There are several basic objects that are needed to do any remote processing. You need to know how to execute remote commands, what machines to run on, how to execute commands that

run on multiple machines at once, and how to send messages between the master process and remote processes.

The `SEP.rc` is the simplest of these build blocks. It defines two variables, `SEP.rc.shell` and `SEP.rc.cp`, which is the shell for a remote process and the copy command. It defaults to using `rsh` and `rcp` for these variables but can be set to using the secure alternatives. It also provides the functions `cp_to(mach,file_in,file_out)` and `cp_from(mach,file_in,file_out)` which return the command strings needed for transferring a file.

The `SEP.mach_base.mach` provides the framework for keeping track of what machines are available. It inherits from the `SEP.stat_sep.status` object to store its current state. It provides a mechanism for testing whether a node is functional. It requires that its children provide a mechanism to create an initial list of machines to run on. It identifies each processor on a machine through a *machine label*, which takes the form `mach-X`, where `mach` is the machine name and `X` is a processor number associated with that machine. The child classes `SEP.mach_file.mach` and `SEP.mach_list.mach` are the simplest two examples, which read their list of available machines from a file or from a supplied list. A future module might interact with a master server allowing a job to shrink or grow based on current computer usage.

In an environment where a master node isn't exporting a disk, or when you don't want to rely on that master node being up, it is necessary to copy a program to all the slave nodes. The class `SEP.distribute_prog.distribute` provides a mechanism to distribute a given program to these nodes. It copies an executable to the `/tmp` directory with a unique name, and returns that name to the calling program. It also has a cleaning method to remove the program from the nodes.

The class `SEP.pc_base.communicator` provides framework for running a job on multiple machines simultaneously. It is initialized with the speed of the network. It expects its children to override the function `prep_run` which defines how to run a parallel job given the list of nodes, the command line arguments, and how many bytes are going to be processed. The last argument is used as a mechanism to calculate how long a job should take, therefore a mechanism to test whether a job is hung and should be killed. The class `SEP.pc_mpich.communicator` is the only current example. It uses MPICH as its communication model.

Communicating with a series of remote processes can be a tricky proposition. The standard Unix approach, a socket, has an important limitation in that it can not have more than X, where X is a small number, of processes waiting to establish a connection. This limitation can be reached either by having to many processes talking on a given socket or by the actions brought on by the socket communication taking too long. The library accounts for both these limitations. The basic concept is that a parallel job might spawn several sockets simultaneously. Each socket will communicate with a maximum number of processes (60 by default). The actions taken after receiving a message will be limited writing to a text file. The class `SEP.par_msg.msg_object` has the ability to read and write a message. Its child, `SEP.par_msg.server_msg_obj`, receives the message over a socket using `SEP.sep_socket.sep_server` class.

Parallel files

As mentioned at the beginning of this section we are going to be running a series of remote processes with local versions of SEPlib files. The parallel file objects control these local versions. They store not only the names of the files but also handle distributing and collecting the files.

The base parallel class `SEP.pf_base.parfile` inherits from both the `SEP.sepfile.sep_file` and the `SEP.stat_sep.status` classes. It is initialized with *at least* an ASCII description, *name*, the usage of the file *usage* ("INPUT" or "OUTPUT"), and the *tag* that the program uses to access it (e.g. <, data=, >). In addition, it must be either initialized with the list of *tasks*, mentioned above, or have its status loaded from disk using the *load* argument. In addition the user can specify

add Whether we are adding the results contained in the parallel file to a preexisting file.

remove Whether (default) or not the local versions of the parallel file should be removed once the jobs is finished.

collect_it Whether (default) or not an output file should be collected when the parallel job is finished.

The two most important functions are the ability to `collect` all of the portions of the parallel file onto the master node, and `tags` which returns the tag that should be used when accessing a local version of the file. The `tags` function is passed in a dictionary linking *task* with the machine it will be run on. A local name for the section of the data is found or created and, if needed, a local version of the file is distributed to the node. The inheritance chart for the parallel file objects can be seen in Figure 1.

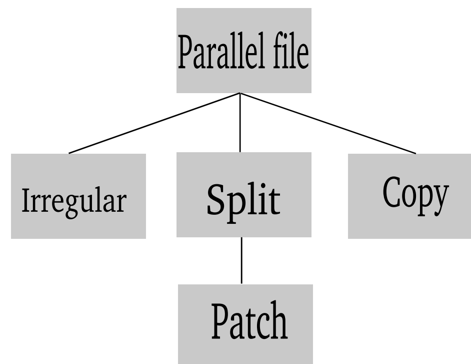

Figure 1: The inheritance chart for the parallel file objects. bob1-parfile [NR]

The class `SEP.pf_copy.parfile` is the simplest of the parallel files. It only works with a regular SEPlib cube. If the named file is an input file, `SEP.pf_copy.parfile` copies the entire file to each node. If the named file is an output file, n collection all of the local version will be summed to produce the final output. This class has the additional initialization argument *reuse_par*, which tells how to signify to the program that the file already exists. This argument

is added to the returned parameters in `tags` when dealing with an output file that already exists. An example of a `SEP.pf_copy.parfile` is the velocity and image files in migration. The *reuse_par* argument would be needed when the image file is already created on the node, to signify that we need to add to rather than replace, the image. Transferring the data is done in megabyte-sized chunks from one node to the next to build up the image using `Copy_join` or to distribute using `Copy_split`.

The class `SEP.pf_split.parfile` allows a parallel file to be split along one or more axes. It is initialized with the additional parameter *dff_axis*, the axis or axes we are splitting along, and *nblock*, the number of portion that axis will be split into. An example of a `SEP.pf_split.parfile` file would be the frequencies in downward-continuation migration. The same principal is used to transfer the data. Megabyte-sized chunks are passed from one node to the next using `Patch_split` and `Patch_join`. When a node contains a given chunk, it is read from or written to disk, otherwise it just passes what it received.

The class `SEP.pf_patch.parfile` inherits from `SEP.pf_split.parfile`. It is used when overlapping patches of a dataset are needed. It adds the additional initialization argument *noverlap*, which tells how much overlap between the patches along each axis. An example would be the image space in shot-profile migration. When collecting, it applies a triangle in the overlapped region.

The last type of parallel file is the class `SEP.pf_irreg.parfile`. It currently can only be used as input. It is defined for irregular datasets (aka SEP3D). It has the ability to be split along multiple axes and have overlapping patches. It use `Sep3d_split` for distribution. The distribution is fairly smart. It does not attempt to sort the dataset but instead makes sure that each local version of the data contains an updated `data_record_number`. As a result, it can be a very effective method to handle large-sized sorts.

PARALLEL JOBS

The controlling process for running a parallel job comes from the `SEP.pj_base.par_job` class or its children. It is derived from the `SEP.opt_base.options` class for parameter handling. There are also numerous optional parameters that can tune the performance on a cluster. There are two required parameters to initialize a parallel job. The first is a dictionary *files* whose values are the parallel files needed for the job. A second is a dictionary *sect_pars* linking tasks to parameters. In addition, most parallel jobs will have *program*, the executable that will be run on each node, and *global_pars*, a list of parameters that each job will need in addition to those described in *sect_pars*. There is a number of other options such as `device` (which tells what Ethernet device the cluster is connected to) that can be useful to tune performance on a given cluster.

At the start of a parallel job, several communication threads are forked. Each of these threads' purpose will be to handle communication between a set of slave processes (the jobs on remote machines) and the master machine. The master thread then requests a list of all of the machines that are available. It checks to make sure each of these machines is functional.

It then begins a loop that runs until each job has run to completion.

The loop begins by requesting from the machine object a list of available *machine labels*. It has to parse this list if any of the parallel files are of type `SEP.pf_copy.parfile` and are being used as output. Only a single process can be started on a given node until the file has been created. It then matches available jobs to the remaining *machine labels*, and requests from each parallel file object a local version of that file. It takes the parameters in *global_pars*, the task parameters in *sect_pars*, and adds in parameters telling the jobs how to communicate with the socket it has been assigned to. Then the command line for a given job is constructed by the *command_func* routine. By default this routine builds the command line based on *program* defined in the initialization. This function can be overwritten for more complex tasks. It forks a new thread for each job, and records that a job has been sent. These forked threads will exit when the job has been completed. If the exit status of the job is not 0, the job will be listed as failing.

Once a series of jobs has been started, the master thread reads the series of files written to by the `SEP.par_msg.server_msg_obj` objects, and updates the status of each job. The status messages come in several forms:

running A task has successfully started. Notification that a job has started successfully is important in the case of an output `SEP.pf_copy.parfile`. The signal is sent when the output file has been successfully created and notifies the server that it is safe to start other jobs on the node.

finished The task has completed successfully. When a job has finished, the machine is marked available. If all jobs are finished the loop is exited.

progress The task has completed a certain portion of its job. If a job is restarted this information is included in the command line options for the job.

failed The task failed. The machine status is checked. If it is no longer working, all jobs that have completed on that node are marked as needing to be rerun. [2] If the node is working, the task is guaranteed to be assigned to another node. If it fails more than twice (also configurable) the job is exited.

The process then sleeps and restarts the loops. Every few minutes it checks to see if any nodes have failed or if any previously failed nodes now work. If the job loop exits successfully, the sockets are shut down and the parallel files are collected as necessary.

There are two extensions to the `SEP.pj_base.par_job` object. The `SEP.pj_simple.par_job` class is all that is needed for most parallel jobs. It takes the additional command line arguments:

command The name of the program to run.

[2] It is possible to tell the parallel job to not rerun these jobs with the assumption that the problems with the node will be fixed.

files The list of files the jobs needs.

tags The list of tags associated with the `files` described above.

usage The usage for each of the files.

nblock The number of parts to break the files into.

axis The axis in which each file is split along.

file_type The file type for each distributed file (*DISTRIBUTE* or *COPY*).

The object then builds all of the appropriate parallel file objects.

The final parallel job class, `SEP.pj_split.par_job`, is useful for many inversion problems. It is initialized with a dictionary `assign_map` linking the job with the machine, or more precisely a machine label, specifying where the jobs should be run. By always running a specific portion of the dataset on a given node, you can avoid collecting the dataset at each step in the inversion process. It can also be useful in things like wave-equation migration velocity analysis where a large file, in the velocity analysis case the wave-field, is needed for calculations. The downside of this approach is, if a node goes down, the job can not run to completion but must terminate when it has accomplished the work on all the remaining nodes.

INVERSION OBJECTS

There are three class trees in the inversion library. The vector class tree defines how to do mathetical functions on a stream of numbers. An operator knows its domain (model space) and range (data space) and how to map a vector from one to the other. Finally, the solver class defines how to estimate a model vector given an operator.

Vector objects

Vectors are simply a stream of numbers that exist in some space. The base vector class is `SEP.vec_base.vector`. The base class defines a series of functions that must be overridden by its children.

clone() Return a copy of the vector.

clone_space() Return a copy of the space the vector exists in.

zero() Zero the vector.

random() Put random numbers into the vector.

scale(val) Scale the vector by the number *val*.

add(vec) Add *vec* to the vector.

scale_add_scale(scale1,vec,scale2) Scale the vector by *scale1* and then add *vec* scaled by *scale2*.

multiply(vec) Multiply the vector by the vector *vec*.

dot(vec) Return the dot product of the vector with the vector *vec*.

load(name) Load the vector *name*.

size() Return the size of the vector.

clean() Clean the vector, remove it from memory and/or disk.

The `SEP.vec_super.vector` class is derived from the `SEP.vec_base.vector` class. It is simply a collection of more than one vector. It applies all mathematical operations on each vector independently.

The `SEP.vec_oc.vector` class is an out-of-core vector that exists in a file. The `SEP.vec_sep.vector` class is inherited from the `SEP.vec_oc.vector` class and the `SEP.sepfile.sep_file` class. It is vectors that are stored in SEPlib files. It uses the SEPlib program `Solver_ops` to perform mathematical operations. The `SEP.vec_oc.vector` should not be used by a programmer because the data type (complex or float) has not been defined. The `SEP.vec_sfloat.vector` and `SEP.vec_scmplx.vector` classes define the float and complex version of the a SEP out-of-core dataset.

There are six additional vector classes for use with parallel jobs. The current inheritance tree for the vector is shown in Figure 2. The `SEP.pv_copy.cmplx_vector` and `SEP.pv_copy.float_vector` classes are for files that are shared among the nodes (derived from the `SEP.pf_copy.parfile`) class and are distributed and collected, before and after each parallel operation. The `SEP.pv_spli.cmplx_vector` and `SEP.pv_spli.float_vector` vectors are also distributed and collected, but are split among the nodes and inherited from `SEP.pf_split.parfile`. The final two vectors, `SEP.pv_always_split.cmplx_vector` and `SEP.pv_always_split.float_vector`, also inherit from the `SEP.pf_split.parfile` class. These two classes are never collected or distributed and exist solely on the nodes. Parallel operators using these vector need to be derived from the `SEP.pj_split.par_job` class. Mathematical operations are done in parallel using the `Solver_ops_split` program.

Operator

The base operator class is `SEP.op_base.operator`. It is initialized with a name (an ASCII string) and by a domain and range vector that are derived from the `SEP.vec_base.vector` class. It can also be initialized with a verbosity flag, *verb*, on whether to print out a message message, *msg* (which defaults to the program name), when applying the forward or adjoint. The forward and adjoint functions require model and data space vectors, which are tested to make sure that they correspond to the domain and range vectors for the operator.

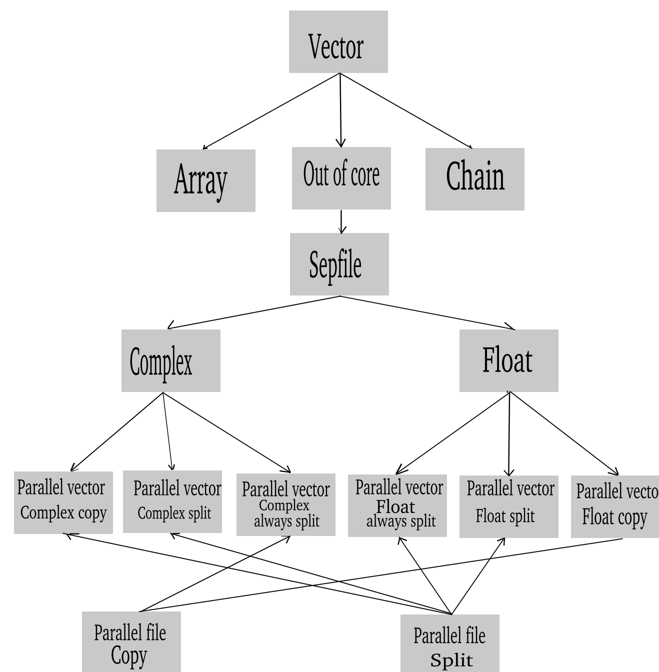

Figure 2: Vector inheritance tree. bob1-vector [NR]

In addition, the forward and adjoint functions have several optional arguments. The *add* is used in to signify that the output of the operation should be added to an existing vector. The *restart* can be used to signify that we are restarting the operator. When the operator is being used an inversion problem, two additional arguments will be passed. The *iter* argument corresponds to the current iteration. The *status* argument is a `SEP.status_sep.status` object used to keep track of the progress of the inversion. If passed in, the starting and finishing of the operation will be recorded in the status file. The forward and adjoint functions only deal with keeping track of the progress of the inversion. The real work is done by the `adjoint_op` and `forward_op` functions. These two functions must be overridden by its children. The `SEP-.oc_base.operator` defines a function `dot_test` which tests to make sure the operator passes the dot product test. The `init_op(restart)` function is defined to perform operations needed before the operator is initialized for the first time. The *restart* argument is used to signify whether the job is being restarted.

The simplest operator that is derived from the `SEP.op_base.operator` class is the `SEP-.op_scale.operator` operator. This operator is simply a diagonal operator where elements along the diagonal are constant. It uses the `SEP.vec_base.vector` vector operations to run the forward and adjoint. It is used by the solver to apply ϵ when doing regularized or preconditioned inversion.

Two classes for combining operator are also derived. The first `SEP.op_combo.chain` class chains two or more operators together (eg. **AB**) , where **A** and **B** are both operators. When initialized, it checks to make sure that the domain of **A** and the range of **B** are the same space. The `init_op` function is overridden to create the temporary vector of the shared space. The `SEP.op_combo.array` class is used to define an array of operators. The number of columns and rows, along with the operators, are passed in during initialization. It can be useful for

regularized problems and building complex inversion operators. The range and domain vectors are constructed from the SEP.op_super.vector vector class.

The current operator tree can be seen in Figure 3. An out-of-core operator class, SEP-.op_oc.operator, is also derived from the SEP.op_base.operator class. This class expects its inputs and outputs will be stored on disk. The SEP.op_oc_serial.operator is used for operators that are applied by a serial code. The class is derived from both the SEP.op_oc.-operator and SEP.opt_par_group.par_group. It is initialized by the location of the serial code *prog* and optionally the *name*, a description of the operator (defaulting to the program name), a verbosity flag (*verb*), and the message to print when applying the operator (*msg*). Other operator's parameters are set using the SEP.opt_par_group.par_group parameter methodology.

domain and range The domain and range vectors. These are expected to be derived from the SEP.op_oc.vector class.

domain_tag and range_tag The tags the program uses for the domain and range vectors, defaults to model= and data=.

restart_com The command line argument to specify a restart, defaults to none.

adj_com The command line argument when running the adjoint, defaults to adj=y.

add_com The command line argument when adding to the output, defaults to add=y.

The forward_op and adjoint_op become calls to the serial code.

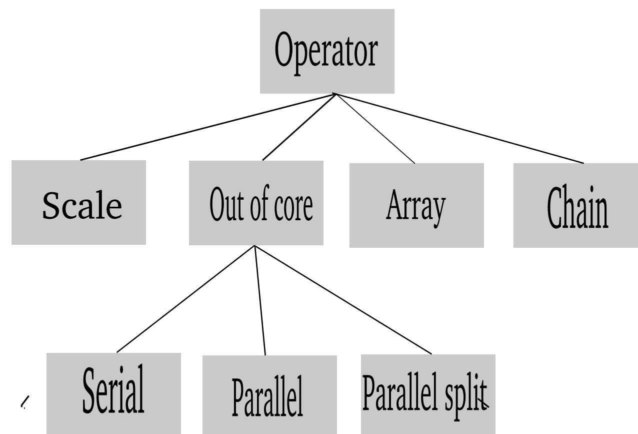

Figure 3: The inheritance class for operators bob1-operator [NR]

The final two operator classes, SEP.op_oc_par.operator and SEP.op_oc_par_split-.operator, are for parallel jobs. They are derived from the SEP.pj_base.par_job and SEP-.pj_split.par_job class along with the SEP.op_oc.operator class. The SEP.op_oc_par.operator class is designed for jobs where the domain and range vectors will be distributed and collected before and after the parallel job. The SEP.op_oc_par_split.operator class is useful

for problems where distributing and collecting the files after every operation is not practical. All of the initialization and parameters are generally handled the same as for the `SEP.op_oc_serial.operator`. The domain and range vectors are either `SEP.pv_copy.vector` or `SEP.pv_split.vector` in the case of `SEP.op_oc_par.operator` operator. In the case of the `SEP.op_oc_par_split.operator` object, the vectors must be of the `SEP.pv_always_split.vector` type. To run the operator, a parallel job is executed rather than running a serial code.

Solvers

The base solver class, `SEP.solv_base.solver`, is inherited from `SEP.opt_base.options`. The solver keeps track of its progress using the `SEP.stat_sep.status` module. It expects that the parameters *niter*, the number of iterations, and *stepper*, the class that calculates the step to be set by the time `prep_run` is called. The function `iter` iterates *niter* times. To make expanding the solver easier, it includes a dummy function `end_iter(iter)` that is run at the end of each iteration. This function can be overwritten to do things like write out the model at each iteration.

There are three classes derived from the `SEP.solv_base.solver` class. The `SEP.solv_smp.solver` class is designed for solving problems of the form

$$Q(\mathbf{m}) = ||\mathbf{d} - \mathbf{L}\mathbf{m}||^2 \quad (1)$$

where $\mathbf{d}$ is the data, $\mathbf{L}$ is the operator, and $\mathbf{m}$ is the model. It expects that its children set the operator *op*, the data *data*, the model *model* before its `prep_run` function is invoked. It will also look for a weighting operator *wop*, a a model vector *v*, an initial model *m0*, and an initial residual vector *resd*. From this information, an initial residual model vector are calculated and `SEP.solv_base.solver's` `prep_run` is invoked.

The `SEP.solv_reg.solver` class is also derived from `SEP.solv_base.solver` class. It is used for regularized inversion problems of the form,

$$Q(\mathbf{m}) = ||\mathbf{d} - \mathbf{L}\mathbf{m}||^2 + \epsilon^2 ||\mathbf{A}\mathbf{m}||^2, \quad (2)$$

where $\mathbf{A}$ is the regularization operator. It expects all of the arguments of `SEP.solv_smp.solver` with the addition of regularization operator *reg* and the relative weighting parameter ϵ *eps*. In addition accepts the optional residual model space vector *resm*.

The final solver derived from `SEP.solv_base.solver` is `SEP.solve_prec.solver`. It is used from regularized problems with model preconditioning of the form,

$$Q(\mathbf{p}) = ||\mathbf{d} - \mathbf{L}\mathbf{B}\mathbf{m}||^2 + \epsilon^2 ||\mathbf{p}||^2, \quad (3)$$

where $\mathbf{B}$ is the preconditioning operator and $\mathbf{p}$ is the preconditioned variable, where $\mathbf{m} = \mathbf{B}\mathbf{p}$. It expects all of the same parameters of `SEP.solv_reg.solver`, with *reg* removed and *prec*, the preconditioning operator, added.

All of the solvers are currently linear solvers. Adding non-linear solvers would be fairly easy. They all require a step function. Currently only a conjugate gradient step function is available in `SEP.cgstep.cgstep`.

PROGRAM FUNCTIONS

There are two required and three optional functions available to the programmer. The required functions go near the beginning and ending of the program. The first, `sep_begin_prog`, tells the server that the program has started successfully. It should be used when it is safe to start another instance of the program on the same node. This latter requirement is important when sharing an output space (`COPY`) on a node. If you are adding to the output space you need insure that the output file has been created by the first instance of the program. The other required function is `sep_end_prog`. This should be used after all of the output files have been completely written to. The server interprets it as a sign of completion for the job.

The `sep_progress` function enables effective job restarting. The programmer can use this function to signify a checkpoint in the code. When restarting a job, both the restart flag and the last progress message will be passed as arguments for the restarting job. In the case when you are sharing an output space, you should use this function whenever you write to the shared file. The final two functions are when you are sharing an output space. The `sep_open_lock(tag)` locks (if already locked waits for the file to become available) a SEPlib tag. The `sep_close_lock(tag)` function frees a tag so another process can safely read or write to it. All of these commands perform no function when they are not a portion of a parallel job.

EXAMPLES

Examples are the most effective way to learn how to use a piece of software. In this section, I will go over several examples. I will begin by showing how to build a flow of multiple programs. I then will give an example of how to run a simple parallel job. The third example will show how to an out-of-core inversion using serial codes, and the final example will show how to do an inversion using parallel objects.

Creating a flow

Creating a wrapper to a serial code involves creating two objects. The first object is a class that knows the parameters for the program and how to execute it. An example is `scale.py` which tells how to run a simple scaling program.

```
import SEP.opt_prog       #handle the parameters for the program
class scale(SEP.opt_prog.options): #inherit from both
  def __init__(self,name="scale"):  #initialization method
    SEP.opt_prog.options.__init__(self,name) #initialize parameters
    self.set_prog("Scale.x")
  def add_doc_params(self):
    """Add the parameters for scaling a dataset"""
    self.add_doc_param("stdout",doc="Output file")
```

```
    self.add_doc_param("scale",doc="Scale the dataset by a given value")
```

The second object will use the above class as in input to a program (`SEP.prog.prog`) object. The script `Scale.py` is an example of using the above `scale` object.

```
#!/usr/bin/env python
import scale        #the scale object code
import SEP.prog     #the program object code

scale=scale.scale("Scale")      #initialize the scale object
program=SEP.prog.prog("Scale.py  ",  #the name of the program for self-doc
   "Scale.py pars ",  #Usage for self-doc
    [scale],  #the components of the program
     ["Scale  using the SEP python library"]) #description doc
program.get_options()        #read the command line arguments
program.prep_run()           #run the program
```

The script functions the same way as normal SEPlib program. If no arguments are given, self-doc is returned. If an argument that is needed isn't present an error is given (in this case including the self-doc for the parameter).

Things get more interesting when we add a second program. In this case, we will add a simple program that will create a 2-D array with a plane of ones at some location. The script `line.py` provides the wrapper for the program.

```
import SEP.opt_prog  #parameters
class line(SEP.opt_prog.options):
  def __init__(self,name="scale"):
    SEP.opt_prog.options.__init__(self,name)
    self.add_prog("Line.x")
  def add_doc_params(self):
    """Add the parameters for scaling a dataset"""
    #add parameters with default values
    self.add_doc_param("stdout",doc="Output file")
    self.add_doc_param("n1",10,doc="The number of sample first axis ")
    self.add_doc_param("o1",0.,doc="Origin of the first axis")
    self.add_doc_param("d1",1.,doc="Sampling of the first axis")
    self.add_doc_param("n2",10,doc="The number of sample second axis ")
    self.add_doc_param("o2",0.,doc="Origin of the second axis")
    self.add_doc_param("d2",1.,doc="Sampling of the second axis")
    self.add_doc_param("sample",5,doc="Sample number at which to create line")
```

We can write a script that combines these two programs. The new program `Line_scale.py` will first create the array with the program line, then use that output as input to the scaling

program. Combining the programs involves three additional steps. First, we will need to create a parameter object, `SEP.opt_prog.options`, that will be responsible for storing the input and output file information. Second, we will need to inherit from both the `line` and `scale` objects. We will change the initialization of these object is to include the new parameter object. We will remove from the documentation the input and output file requests. We will set the input and output file names based on the new parameter object. The final changes involve how to recognize the parameters for the various programs. We will introduce a new dictionary *prefixes* which maps a prefix to the parameter objects. In the example below, you will now use *line_n1* to set the number of samples in the output space.

```
#!/usr/bin/env python
import scale,line       #import both objects
import SEP.opt_prog
import SEP.prog

class main_args(SEP.prog):  #the main parameters
  def __init__(self):
     SEP.opt_prog.options.__init__(self,"MAIN")
     self.add_doc_param("stdout",doc="Output file")

class my_line(line.line):  #inherit from the line object
  def __init__(self,main_pars,name):
    line.line.__init__(self,name)  #initialize the line.line structure
    self.main=main_pars   #store the main programs parameter class
  def add_doc_params(self):
    line.line.add_doc_params(self)  #add the parameters described in line
    self.del_par("stdout")a  #delete the line parameter
  def prep_run(self,restart):
    #set the stdout a temp file based on the stdout of the main program
    self.add_param("stdout","%s.temp"%self.main.param("stdout"))
    line.line.prep_run(self,restart)

class my_scale(scale.scale):   #inherit the scale calss
  def __init__(self,main_pars,name):
    scale.scale.__init__(self,name) #initialize the scale class
    self.main=main_pars
  def add_doc_params(self):
    scale.scale.add_doc_params(self)
    self.del_par("stdin") ;self.del_par("stdout") #delete in and out
  def prep_run(self,restart):
    #add in out  (in is temp file from line)
    self.add_param("stdin","%s.inter"%self.main.param("stdout"))
    self.add_param("stdout",self.main.param("stdout"))
    scale.scale.prep_run(self)
```

```
main=main_args()   #create the main arguments
line=my_line(main,"Line")  #create the line arguments
scale=my_scale(main,"Scale")  #create the scale arguments
prefixes={}  #prefix dictionary
prefixes["Line"]="line_"  #prefix for all line parameters
prefixes["Scale"]="scale_" #prefix for all scale parameters
program=SEP.prog.prog("Scale_line  ",
   "Scale_line.py pars  outtag= ",
    [line,scale],  #now we have two objects in the flow
     ["Create a line and then scale it  using the SEP python library"],
     prefixes=prefixes  #the prefixes associated with the objects
   )

program.get_options() #get the options
program.prep_run()     #run the flow
```

Simple parallel jobs

For many parallel jobs, the program `Parallel` is all that is needed. `Parallel` is meant for parallel jobs where the input and output are either distributed, `SEP.pf_splist.parfile`, or share input, `SEP.pf_copy.parfile`, and the data is partitioned along a single axis for each file. It also requires that you are running a single program on each node (rather than some more complex operation) and you aren't wanting to add the output to another file. The required arguments to `Parallel` are composed of the program name, *command*, the number of blocks to break the program into *nblock*, and a series of lists (comma separated).

files The name of the parallel file(s).

tags The tags associated with each file.

axis The axis that each file is split along. If the file is shared, the axis is ignored for this file.

usage The usage for each file "INPUT" or "OUTPUT".

file_type The parallel file type for each file ("DISTRIBUTE" or "COPY").

All arguments that aren't part of the `Parallel` program are passed as command line arguments to parallelized serial code.

The program is effective for parallelizing code where the computational cost is significantly more than the cost of transferring the data (migration and modeling for example). It is also effective when handling problems that benefit from being held in memory (operations such as transposes). For example, a multi-gigabyte 2-D file could be transposed at marginally more than the cost of distributing and collecting the dataset through

```
Parallel files="in.H,out.H" tags="stdin,stdout" axis="2,1" usage="INPUT,OUTPUT"\
  file_type="DISTRIBUTE,DISTRIBUTE"
```

Complex parallel job

The script `fdmod.py` in the report directory parallelizes 2-D finite difference modeling using the SEPlib program `Fdmod`. The general form is the same as seen in the scripts in the flow example. The `fdmod_par` object extends the `SEP.pj_base.par_job` class. The user describes a regular spaced set of shot locations using the parametes *oxs,dxs,nxs,ozs* and the number different blocks to break the problem into using *nblock*. Each job process a different set of shot locations and needs its own set of parameters. The function `build_sect_pars` calculates the parameters that vary as a function of job and stores the contents in the dictionary *sect_pars*.

```
  def build_sect_params(self):
    """Build parameters for parallel job"""
    sect_pars={}
    nxs=int(self.param("nxs"))
    oxs=float(self.param("oxs"))
    dxs=float(self.param("dxs"))
    ozs=float(self.param("ozs"))
    nblock=self.param("nblock")
    if not nblock: nblock=nxs
    nblock=int(block)
    imin=int(n/nblock)
    nextra=n-nblock*imin
    itot=0
    for i in range(nblock):
      sect_pars[str(i)]=SEP.args.basic(name=str(i))
      ol=o+d*itot
      dl=d
      nl=imin
      if i < nextra: nl=nl+1
      itot=itot+nl
      sect_pars[str(i)].add_string("nzs=1 ozs=%f dzs=1."%(ozs))
      sect_pars[str(i)].add_string("nxs=%d oxs=%f dxs=%f"%(nl,ol,dl))
    return sect_pars
```

The `prep_run` function creates the list of parameters and defines a dictionary of parallel files. The velocity space is copied to all of the nodes using the `SEP.pf_copy.parfile` class, the output shot gather files are spread accross the cluster using the `SEP.pf_split.parfile` `object`.

```
   par_files={}
```

```
   par_files["vel"]=SEP.pf_copy.parfile(name=self.param("intag"),
     tag="intag=",usage="INPUT",njobs=len(sect_pars.keys()),restart=restart)

   par_files["hsfile"]=SEP.pf_split.parfile(name=self.param("hsfile"),
    dff_axis=3,tag="hsfile=",usage="OUTPUT",njobs=len(sect_pars.keys()),
     restart=restart,nblock=len(sect_pars.keys()))
```

The section parameters, the parallel files, and the modeling program are then added to the objects parameters and the parallel job is initialized.

```
   self.add_param("files",par_files)
   self.add_param("sect_pars",sect_pars)
   self.add_param("program","%s/Fdmod"%SEP.paths.sepbindir)
   SEP.pj_base.par_job.prep_run(self)
```

Inversion example

The script `Interp.py` does out-of-core interpolation. The program `Interp.x` interpolates from an irregular to a regular mesh. The program `reg.x` convolves with a three point filter. We then can turn these program into operators.

```
interp_op=SEP.op_oc_serial.operator("Interp.x")
reg      =SEP.op_oc_serial.operator("reg.x")
```

We then extend the solver object. We require the operator *op* and regularization operator *reg*. (*n1*,*o1*,*d1*).

```
class solver(SEP.solv_reg.solver):
  def __init__(self,op,reg):
    SEP.solv_reg.solver.__init__("SOLVER")
    self.add_param("op",op)
    self.add_param("reg",reg)
```

We require the user to specify the model *model*, the data *data*, ϵ *eps*, and the number of iterations *niter*, and the dimensions of the output space

```
  def add_doc_params(self):
    self.add_doc_param("eps",1.,"Epsilon")
    self.add_doc_param("niter",1,"Number of iterations")
    self.add_doc_param("model",doc="Model (output)")
    self.add_doc_param("data",doc="Data (input)")
    self.add_doc_param("n1",doc="number of samples (axis 1)")
    self.add_doc_param("o1",doc="First sample (axis 1)")
    self.add_doc_param("d1",doc="Sampling  (axis 1)")
```

We create the data vector from a file and create the model vector.

```
  def prep_run(self,restart=None):
    self.add_param("data",SEP.vec_sfloat.vector(tag="data"))
    self.add_param("model",SEP.vec_sfloat.vector(name="model"))
    self.param("model").set_axis(1, self.param("n1").
      self.param("o1",self.param("d1")))
    self.param("model").zero()
```

We add the domain and range vectors for the operator and the regularization operator.

```
    self.param("op").add_param("domain",self.main.param("model"))
    self.param("op").add_param("range",self.main.param("data"))
    self.param("reg").add_param("domain",self.main.param("model"))
    self.param("reg").add_param("range",self.main.param("model"))
```

Finally we create the step operator and run the `prep_run` function for the operators and the regularization solver.

```
    self.add_param("cgstep",SEP.cgstep.cgstep("cgstep"))
    self.param("op").prep_run(restart)
    self.param("reg").prep_run(restart)
    SEP.solv_reg.solver.prep_run(self,restart)
```

We create the solver object, the program object, read the parameters, and run the job.

```
solv=solver(interp_op,reg_op)

program=SEP.opt_prog.prog("Interpolate  ",
   "Interp.py model= data= ",
    [interp_op,reg_op,solv],
     ["Interpolatation  using the SEP python library"])
program.get_options()
program.prep_run()
```

CONCLUSIONS

In this paper I present simple serial codes that can be used to create much more complex objects with minimal additional coding overhead. These objects can take the form of complex program chains. In addition the serial codes can be used to create coarse-grained parallel programs. A further option is to use simple serial codes in a parallel out-of-core inversion, using both serial and parallel operators. Examples demonstrate how to accomplish each of these goals.

REFERENCES

Clapp, M. L., and Clapp, R. G., 2005, 3-d subsalt imaging via regularized inversion with model preconditioning: SEP–**120**, 1–22.

Clapp, M. L., 2003, Directions in 3-D imaging - Strike, dip, both?: SEP–**113**, 363–368.

Clapp, R. G., 2004a, Fault tolerant parallel SEPlib: SEP–**117**, 175–182.

Clapp, R. G., 2004b, A python solver for out of core, fault tolerant inversion: SEP–**117**, 183–190.

Sava, P., and Biondi, B., 2003, Wave-equation MVA: Born Rytov and beyond: SEP–**114**, 83–94.