Modaldecomp Guide

Brandt Belson

July 17, 2011

1 Overview

1.1 Motivation

The modaldecomp library finds decompositions of fields into primary modes. It implements several methods for doing this, including the Proper Orthogonal Decomposition (POD) [1], balanced POD (BPOD) [1], and the Dynamic Mode Decomposition (DMD) []. These modes are useful for analyzing important features in the original set of fields, and for coming up with reduced-order dynamical models. The most common case in our group is to find the modal decomposition of time-sampled snapshots of fluid simulation velocity fields, however the library is intended to be general and work for any set of data.

Prior to modaldecomp, every member of our lab wrote their own implementations of these algorithms. It usually took awhile for each person to test their work and be confident in the results. Additionally, they tended to not be well optimized for computational efficiency. This library will save new students from writing their own versions, instead only having to interface to modaldecomp. The classes and functions provided benefit from being heavily tested with unittests, and by current and past users. The library is designed to handle small and large datasets (where only a subset of the data can be in memory simultaneously) computationally efficiently.

1.2 Basics

The library is currently located in a hg repository on sequoia.princeton.edu and can be cloned with

```
hg clone ssh://sequoia.princeton.edu//home/repositories/modaldecomp
    mymodaldecomp
```

The library itself is written in python and is parallelized with distributed memory via the module mpi4py, which is available here (installation instructions for the lab macs are on the group wiki). The only required dependencies are Python 2.7 and numpy. To run in parallel, MPICH2 and mpi4py are required, however it can be run in serial without these.

The documentation uses sphinx (available here or use "easy_install sphinx"), and can be generated (if not provided) and viewed from the modaldecomp directory with the commands below.

```
sphinx-build doc doc/_build
open doc/_build/index.html
```

This should open your default browser with documentation taken from the doc strings in the python code. If the second step doesn't work, you can also navigate to `modaldecomp/doc/_build` and open index.html with a browser (e.g. firefox).

To run the tests, use:

```
./runalltests
```

Note that some of these tests are run in parallel. If you don't have mpi4py installed, you can safely ignore any errors from the parallel tests. The way we run all tests might change in the future to use automatic test discovery.

2 User Interface

The library is written to handle *any generic field object*; it is intentionally extremely flexible so it can fill many different users' needs. That means your data can be simple n-dimensional numpy arrays, your own class, or any other data type which satisfies the requirements. Using the code requires writing a few python functions which manipulate your data, generally "snapshots" - time-sampled fields, and passing them as arguments to the constructors of the main classes like POD, BPOD, and DMD. The functions must follow the restrictions below.

2.1 Required functions

1. `save_field(fieldObject, fieldPath)`

   The function `save_field` takes two arguments. The first is the field object (again, this can be any object, like a numpy array or your own class). The second is a string containing the path to the file, for example `"../data/snapshot02.txt"`. Typically one writes save_field so that it saves fields in same the format as an existing simulation, or so that the snapshots can be easily visualized in Matlab, Tecplot, VisIt, etc. If the simulations are done in another language like C/C++ or Fortran, you may already have a function which saves. It is possible to use your existing functions since Python can wrap these languages, and many others. See the discussion below about when it is a good idea to wrap another language.

2. `fieldObject = load_field(fieldPath)`

   The function `load_field` takes the path to a file (usually a string, e.g. `"../data/snapshot02.txt"`) and returns a field object. Typically this function loads files from an existing simulation, such as snapshot files. This function will be called many times, so if you have many fields and speed matters you should make it fast (more on speed in a later section). It is possible to wrap existing load functions from other languages, see the discussion below.

3. `innerProduct = inner_product(fieldObject1, fieldObject2)`

   This function takes the inner product of two field objects, and returns it. If you're unsure about why there are inner products, see the math section.

2.1.1 Wrapping other languages

It can take some time to do this, so I'd recommend trying it only if some of these apply to you:

- You have experience wrapping other languages with python.
- Speed is vital and you're concerned Python isn't fast enough. Python and numpy can be fast for large array manipulations, so don't just assume Python will be slow.

- The function involves complicated operations on the data that you don't want to re-code.
- You have an inherited simulation legacy code and trying to make sense out of the functions is hard. If so, I feel your pain(!), and sometimes it's simply not worth the headache of making sense out of old code. However, keep in mind that it might still be easier and more beneficial to learn what your inherited code is doing and rewrite it in an organized way using high-level Python and numpy. You would probably reuse your rewritten functions for something else too.

If you want to try wrapping another language, see SWIG and f2py.

2.2 Field object requirements

In addition to the required functions which operate on your field objects, the field objects themselves must also satisfy a few properties.

1. Field addition - It must be possible to add two field objects with the "+" operator. For user-defined classes, define a method `__add__(self, other)` for your class (`http://docs.python.org/reference/datamodel.html`documentation).
2. Scalar multiplication - It must be possible to multiply a field object by a number with the "*" operator. For user-defined classes, define a method `__mul__(self, other)` for your class (`http://docs.python.org/reference/datamodel.html`documentation).

Note that numpy arrays already have these operators defined.

2.2.1 Check you meet requirements

To check that your field objects behave in the required way, we supply you with a useful function:

```
fieldoperations.FieldOperations.idiot_check(testObj=None,
   testObjPath=None)
```

Create an instance of FieldOperations, and call the method idiot_check.

```
import fieldoperations as FO
myFO = FO.FieldOperations(inner_product = inner_product, load_field
    = load_field)
myFO.idiot_check(testObjPath="../data/snapshot02.txt")
```

That's it! As long as you define the three functions and the field objects satisfy the two requirements, you can use all of the modaldecomp library.

2.3 Basic Usage

The most detailed usage documentation is the sphinx-generated html (described before). Just for the basic idea of the structure of a script that uses modaldecomp, here is a rough outline. In this example, the user defines their own field class. This is not necessary, one could simply write functions which load, save, and take inner products of numpy arrays.

-- mainBPOD.py --

```
import bpod

class Field(object):
    def __init__(self):
        pass

    def load(self, path):
        pass

    def save(self, path):
        pass

    def inner_product(self, other):
        return (self.data*other.data).sum()

    def __mul__(self, other):
        newField = Field()
        newField.data = self.data * other
        return newField

    def __add__(self, other):
        newField = Field()
        newField.data = self.data+other.data
        return newField

# Define your functions
def save_field(field, path):
    field.save(path)

def load_field(path):
    field = Field()
    field.load(path)
    return field

 def inner_product(field1, field2):
    return field1.inner_product(field2)

# Create an instance of the BPOD class.
# maxFieldsPerNode is max number of fields that fit in memory.
# Set it as large as possible for speed, see section on computation
myBPOD = bpod.BPOD(load_field=load_field, save_field=save_field,
    inner_product=inner_product, maxFieldsPerNode=1000)

# Create lists of snapshot (field) names.
numSnaps = 100
directSnapPath = "../data/direct_snap%02d.txt"
directSnapPaths = [directSnapPath%i for i in xrange(numSnaps)]

adjointSnapPath = "../data/adjoint_snap%02d.txt"
adjointSnapPaths = [adjointSnapPath%i for i in xrange(numSnaps)]
```

```
# Compute the decomposition (forms Y'X and takes its SVD)
myBPOD.compute_decomp(directSnapPaths, adjointSnapPaths)

# Save the Y'X matrix, and the decomposition matrices
myBPOD.save_hankel_mat(hankelPath)
myBPOD.save_decomp(LSingVecsPath, singValsPath, RSingVecsPath)

# Compute and save the modes.
# Note the space for a number %03d, this must be done.
# The modes are also field objects.
myBPOD.compute_direct_modes(range(1,r), '../data/direct_mode_%03d.txt')
myBPOD.compute_adjoint_modes(range(1,r), '../data/adjoint_mode_%03d.txt')
```

To run this script in serial use the line below.

```
python mainBPOD.py
```

2.4 Running in parallel

Runing in parallel requires the mpi4py module, and is done with the command below.

```
mpiexec -n <number of processors> python mainBPOD.py
```

It's simple, nothing needs to be changed in the main script because all of the parallelization is handled internally.

When using modaldecomp in parallel, there are a few things to keep in mind. Modaldecomp makes use of all available processors from the mpiexec command. All of the processors will be reading and writing from the same hard drive, and sometimes using more processors actually hurts speed because the hard drive cannot efficiently read/write to different locations efficiently. This depends on the system, so the modaldecomp library does not optimize this. If you suspect this effect is occurring, lower the number of processors/node. For example, if you are submitting a job on a cluster, keep the number of nodes the same, but only change the "processors per node", sometimes called "ppn". If you're running on a workstation, there is only one node, so lower the number of processors.

Modaldecomp has only distributed-memory parallelization (MPI), treating each processor as if it has its own memory. However, there are gains to be made by using shared memory within a node, and this can still be achieved. To do this, run modaldecomp with only one processor per node, but have all of the other processors available (exactly how to do this depends on the system used). Then make the functions passed to the modaldecomp classes use *all of the processors within a node*. For example, inner products can be taken using all the processors within each node with numpy built with Intel MKL (MKL may do this automatically). While we haven't tested this setup, it seems like it would be the most efficient way to use modaldecomp because it reduces the amount of internode communication and could avoid simultaneous read/writes to disk.

For more details, see the sections on implementation and previous parallelizations, or read the sphinx documentation of the `FieldOperations` class.

3 Mathematical background

For the full derivations and other details about the modal decompositions, see the original publications. Here we only describe how we generalize these decompositions. The original publications tend to write

the equations in terms of matrix operations, where snapshots are stacked into columns of matrices. We take an equivalent, but different point of view. Instead of stacked field (e.g. snapshot) matrices, the elements of our matrices are fields themselves rather than the elements of the fields, and matrix multiplication is generalized to inner products. This will become clearer in the following equations.

All of the decompositions take a set of fields and find their important modes. The definition of "important" depends on the type of decomposition (e.g. POD is by energy, BPOD is by input-output dynamics, DMD by frequency content(?)). The modes are just a linear combination of the fields. In this sense, all that needs to be done is to compute the coefficients so that:

$$\text{mode}_{j=1,r} = \sum_{i=1}^{m} \text{field}_i * a_{ij} \tag{1}$$

where a_{ij} are the coefficients, r is the number of modes, and m is the number of fields. This equation can be written in matrix form.

$$\begin{bmatrix} \text{mode}_1 & \text{mode}_2 & \dots & \text{mode}_r \end{bmatrix} = \begin{bmatrix} \text{field}_1 & \text{field}_2 & \dots & \text{field}_m \end{bmatrix} * A \tag{2}$$

where A is the matrix composed of coefficients a_{ij} with m rows and r columns. The decomposition problem is now a matter of finding the A matrix with entries a_{ij}. This is done differently for each type of decomposition.

3.1 POD

ă

Again, skipping the details, to find the POD of a set of fields the operations are described in [1] as:

1. Collect data and store in one-dimensional column-vectors $x_{i=1,m}$, where each x_i vector has n entries.
2. Stack the data into a matrix $X = \begin{bmatrix} x_1 & x_2 & \dots & x_m \end{bmatrix}$, so X is n by m.
3. Take the SVD of X^*X, giving $UEV^* = X^*X$, only keeping the non-zero singular values and corresponding first columns of U and first rows of V^*.
4. The modes are then the rows of the matrix $\phi = XVE^{-1/2}$, and are ranked by energy content.

In our point of view, the formation of the X matrix is never done explicitly. Instead, notice that each entry (i,j) in the product X^*X is actually the inner product of x_i with x_j. That is, $(X^*X)_{ij} = \langle x_i, x_j \rangle$. In fact, the inner product notation is more general. Simply multiplying the x vectors and summing the result assumes a particular inner product that is is not always valid. For example, if the data is on a non-uniform grid then each point has an associated weight. Similarly, if the data doesn't represent physical values at grid points but instead represents coefficients of spectral functions (like Fourier modes), the inner products are different.

The approach taken in this library is:

1. Collect data and store fields $\text{field}_{i=1,m}$. Typically one should store the fields by saving them to file. There is no need to make your data into column vectors.
2. Form $H = X^*X$, by taking all of the inner products $\langle \text{field}_{i=1,m}, \text{field}_{j=1,m} \rangle = H_{ij}$.
3. Take the SVD of $H = X^*X$, giving $UEV^* = H$, only keeping the non-zero singular values and corresponding first columns of U and first rows of V^*.

4. Construct the A matrix of Equation 2, $A = VE^{-1/2}$. The modes, ranked by energy, are found by the linear combinations in Equation 2 (in practice we use Equation 1 for computational efficiency).

3.2 BPOD

Our approach to BPOD is very similar to POD, so for more explanation see the POD section. In [1] the BPOD is explained and the steps to find the modes are described as:

1. Collect direct and adjoint data and store in one-dimensional column-vectors $x_{i=1,m_d}$ and $y_{i=1,m_a}$, where each x and y vector has n entries.

2. Stack the data into a matrices:

   $X = [\ x_1 \quad x_2 \quad \dots \quad x_{m_d}\ ]$, so X is n by m_d.

   $Y = [\ y_1 \quad y_2 \quad \dots \quad y_{m_a}\ ]$, so Y is n by m_a.

3. Take the SVD of the Hankel matrix Y^*X, giving $UEV^* = Y^*X$, only keeping the non-zero singular values and corresponding first columns of U and first rows of V^*.

4. The direct modes are then the rows of the matrix $\phi = XVE^{-1/2}$. The adjoint modes are then the columns of the matrix $\psi = E^{-1/2}U^*Y^*$. Both sets of modes are ranked by how controllable and observable they are – how important they are to the input-output dynamics.

The approach taken in this library is:

1. Collect data and store fields direct_field$_{i=1,m_d}$ and adjoint_field$_{i=1,m_a}$. Typically one should store the fields by saving them to file. There is no need to make your data column vectors.

2. Form $H = Y^*X$, by taking all of the inner products $\langle \text{adjoint\_field}_{i=1,m_a}, \text{direct\_field}_{j=1,m_d} \rangle = H_{ij}$.

3. Take the SVD of $H = Y^*X$, giving $UEV^* = H$, only keeping the non-zero singular values and corresponding first columns of U and first rows of V^*.

4. Construct the A matrix of Equation 2 for the direct modes, $A = VE^{-1/2}$ and for the adjoint modes, $A = UE^{-1/2}$. The direct and adjoint modes are found by the linear combinations in Equation 2 with the corresponding (direct or adjoint) set of fields and corresponding A matrix (in practice we use Equation 1 for computational efficiency). The modes are ranked by observability and controllability.

3.3 DMD

DMD can be expressed in way that is very similar to POD. For details, ask Jonathan Tu, jhtu@princeton.edu.

4 Implementation and Computation

As mentioned before, modaldecomp is designed to handle many different types and sizes of datasets easily and efficiently. In this section, we summarize the implementation decisions and mention how they effect efficiency. Most users could probably ignore most of this section. For even more details, see the sphinx documentation

4.1 FieldOperations class

The low-level class FieldOperations (in fieldoperations.py) is the backbone of modaldecomp. The POD, BPOD, and DMD classes all have an instance of FieldOperations as a datamember "slave" that does the heavy-lifting. It need not be used by a typical user, but for a specific need not filled by the higher level classes, it can be used. This class also contains all of the non-trivial parallelization. For that reason, you will rarely have to write any code that requires thinking about whether or not you are in parallel.

There are two primary groups of functions in FieldOperations, those to compute inner product matrices such as X^*X and Y^*X, and those to compute linear combinations of fields, useful for finding the modes from the original fields in Equation 1.

4.2 Inner product matrices

The first of this group is `compute_inner_product_mat`, and it computes all of the inner products between two different sets of fields. This is useful for BPOD, where the sets of fields are the direct and adjoint snapshots. It does this in parallel and in memory-efficient chunks. Since not all of the fields can be in memory at once, only a subset, what we call a chunk, of the "row fields" and a chunk of "column fields" are loaded simultaneously. All of the inner products that can be computed between these two subsets of fields are computed. Then, if in parallel, the subset of column fields are sent to the processor with one higher rank, and a new chunk of inner products are computed. Figures 1 and 2 show how the inner product matrix is divided into chunks.

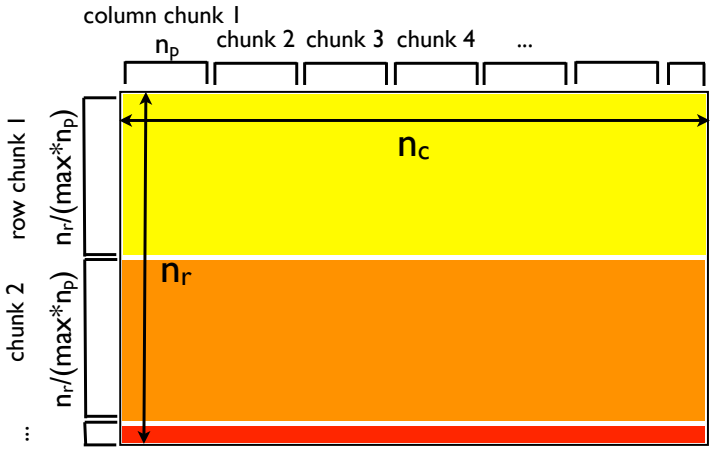

Figure 1 Overview of inner product matrix parallelization.

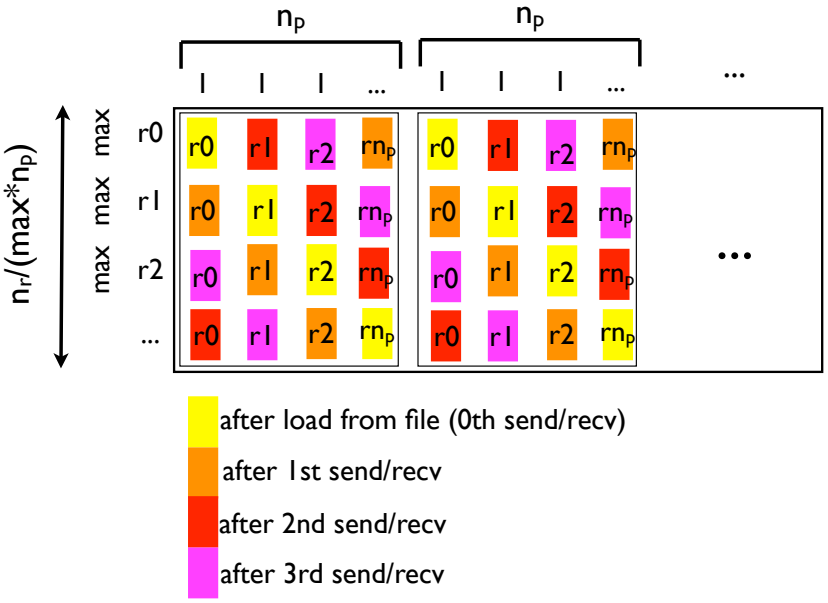

Figure 2 Individual row chunk of inner product matrix parallelization.

This procedure scales the following way:

- number of loads / processor = $\frac{n_r}{max*n_p}\frac{n_c}{n_p} + \frac{n_r}{n_p}$
- number of MPI send-receive pairs / processor = $\frac{n_r}{max*n_p}\frac{(n_p-1)n_c}{n_p}$
- number of inner products / processor = $\frac{n_r}{n_p}n_c$

Where n_p is the number of processors, max is the maximum number of fields per nodes minus one, n_r is the number of rows, and n_c is the number of columns. To show the scaling, we ran the benchmark.py file with a large number of random fields, each containing about 8,000 numbers and computed the inner product matrix with a varying number of processors and nodes. The results are shown in Figures 3, 4, ...

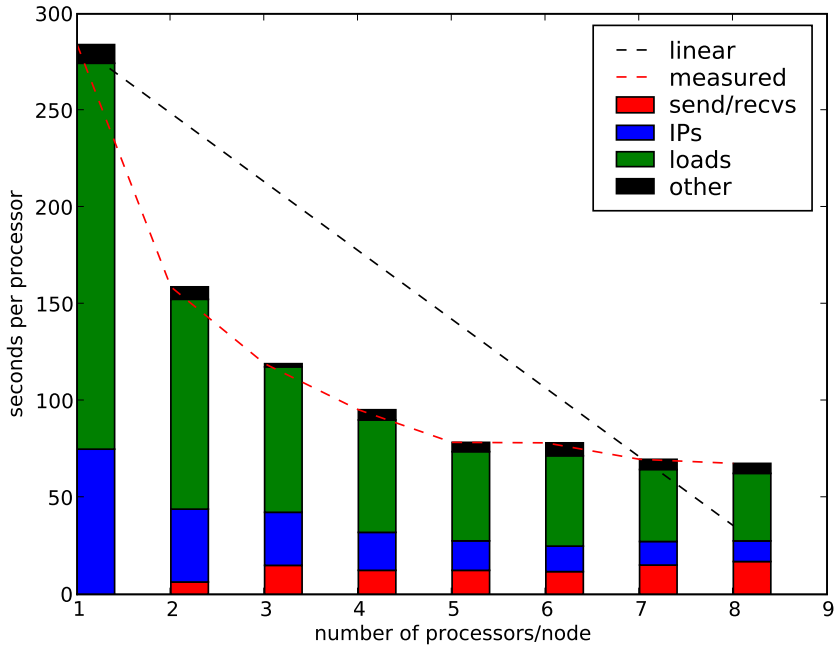

Figure 3 Measured scaling on 8-core Mac Pro (rainier).

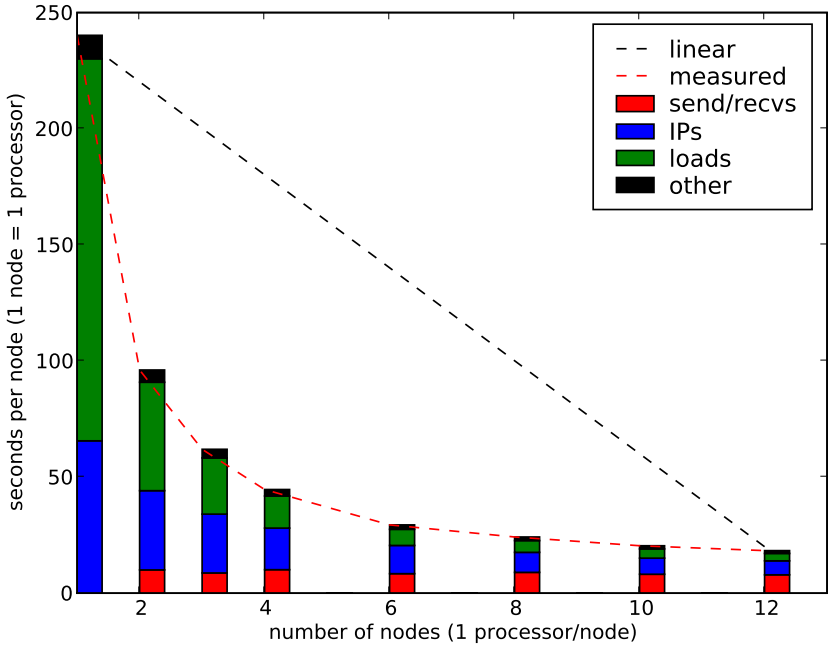

Figure 4 Measured scaling on Della compute nodes, using only 1 processor per node.

The second of this group assumes the two sets of fields are the same, as is the case for POD,so there

are theoretically half as many inner products to compute. A different method is used, and is implemented in compute_symmetric_inner_product_mat. Currently this is not complete, neither is DMD. POD and DMD still work, but they use an old method which has poor scaling for multiple processors/node. When using POD and DMD right now, use only one processor per node. Jon will fix this soon, ask him about it jhtu@princeton.edu.

4.3 Linear combinations

The second group of functions computes the modes from the original fields, as in Equations 1 and 2. All of the decompositions use this method. The basic approach is that each processor is responsible for a subset, or chunk, of the sum fields (on the LHS of Equation 1). Each processor reads a set of the basis fields (on the RHS of the equation) multiplies them by the corresponding $a_i j$ coefficients, and adds this contribution to the sum fields. We call the contribution from a chunk of basis fields to a chunk of sum fields a "layer" because when you sum all of the layers together, you get the full set of sum fields. After each processor computes the layer from the chunk of basis fields, if in parallel, the basis fields are passed to the processor with rank one greater, and the process is repeated. If there were several chunks of sum fields, then this process is repeated for each chunk of sum fields. Figures 5 and 6 show how the sum fields and basis fields are divided into chunks.

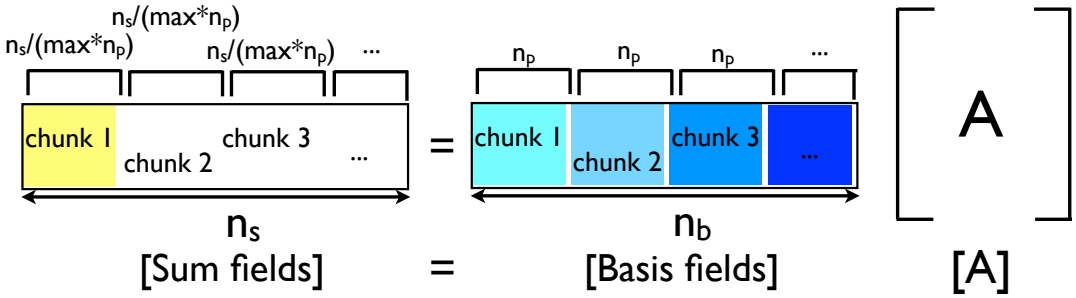

Figure 5 Overview of linear combination parallelization.

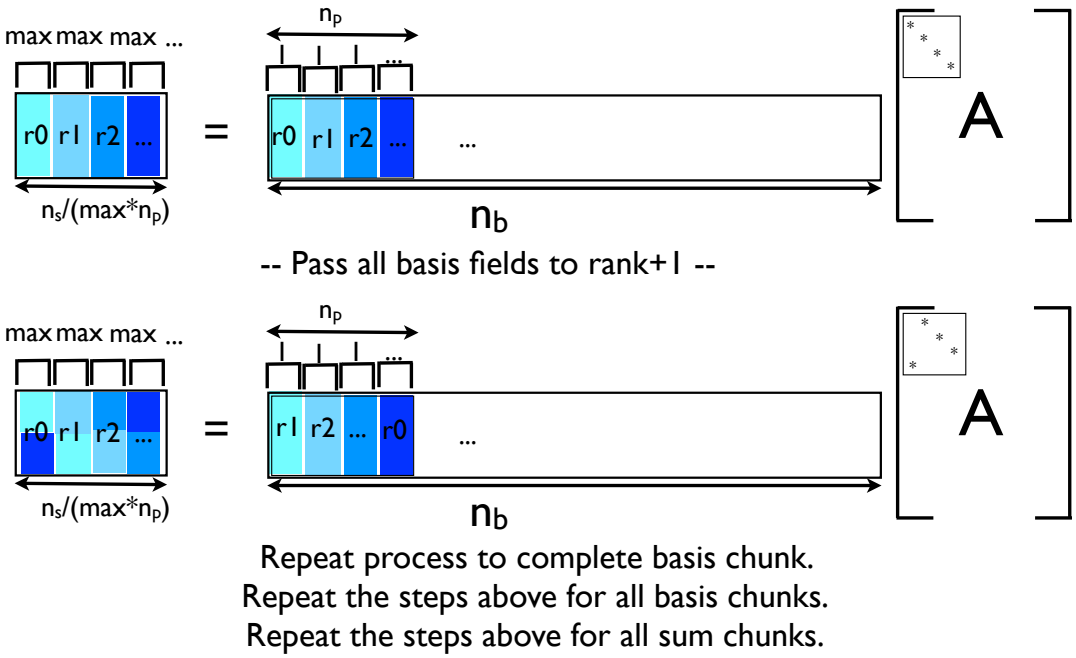

Figure 6 Individual sum chunk of linear combination parallelization.

This procedure scales the following way.

- number of loads / processor = $\frac{n_s}{max*n_p}\frac{n_b}{n_p}$
- number of MPI send-receive pairs / processor = $\frac{n_x}{max*n_p}\frac{(n_p-1)n_b}{n_p}$
- number of scalar multiply, field addition pairs / processor = $\frac{n_s}{n_p}n_b$

where n_s is the number of sum fields, n_b is the number of basis fields, n_p is the number of processors, and *max* is the maximum number of fields per nodes minus one (the same as in the inner product matrix calculations).

The scaling for a similar data set as for the inner product mat was tested with benchmark.py and the results are shown in Figures 7 and **??**.

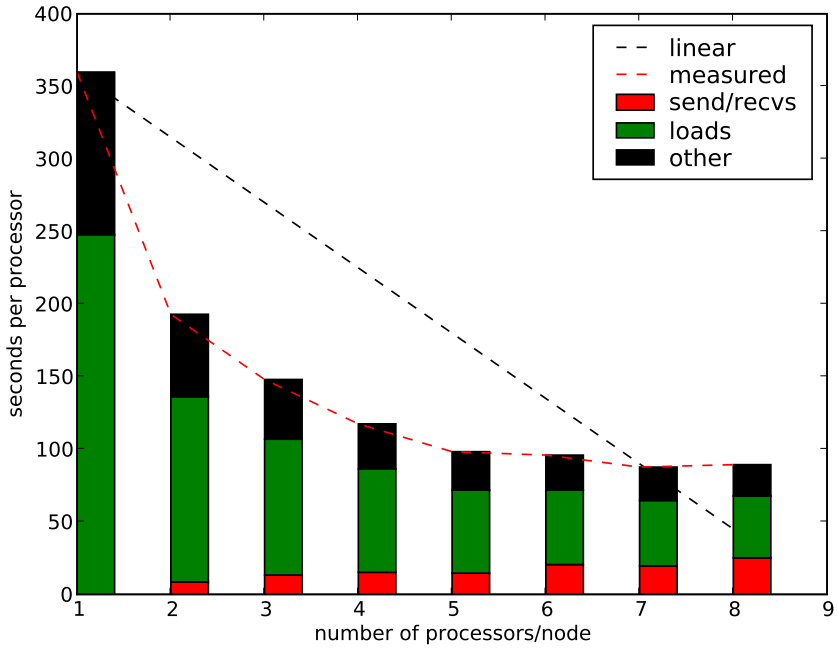

Figure 7 Measured scaling on 8-core Mac Pro (rainier).

5 Previous parallelizations

We've tried different schemes in the past, and they are listed here for future reference. You can ignore this entire section if you're not planning on working on the guts of modaldecomp's parallelization.

5.1 Original distributed version

The original scheme was also a pure-distributed (MPI) parallelization. However, rather than sending fields which were loaded by one processor to another processor, we reloaded. Loading typically is one of the more computational expensive parts of modaldecomp (it depends on the user-provided `load_field`), while MPI send and receives are generally cheaper, so we switched to using send and receives when possible. The old method had poor scaling with respect to the number of loads due to the repeated loading. In fact, when loading was relatively expensive (as it generally is), it actually hurt performance to use more than one processor per node because the memory, specifically the maximum number of fields that could be in memory per node, had to be divided among all the processors, and more reloads occured.

5.2 Shared memory version

To solve this problem, we thought using shared memory (and the built-in Python multiprocessing module) could improve performance because it would eliminate the need to reload some fields entirely. In fact, fields wouldn't even have to be sent with MPI send and receives since the fields are accessible to all processors within a node. In this way, shared memory still could be useful in modaldecomp. Unfortunately we had several major problems, some of these could be solved in the future.

The first was that loading with all processors on a node can be slower than loading with only one processor because the hard drive can only be accessed at one point at a time, and it must jump around to read multiple files simultaneously. This is not true for high-performance hard-drives, but it seems to be on workstations (like the Mac Pros and iMacs). This is also a problem, although to a lesser extent, for the current distributed version though when there is more than one processor per node.

A second problem was that numpy operations were significantly slower when done on multiple processors simultaneously. We noticed that it was faster to complete a set of inner products, which use numpy array multiplication, when done one-by-one on a single processor than when multiple inner products were computed simultaneously with the multiprocessing module's Pool.map() function. We emailed the numpy mailing list about this, but no one actually answered why numpy would be slower for operations done with multiprocessing.Pool.map(). A possibility is that numpy is already using many threads on all of the available processors, so when using multiprocessing, we are giving the system too much to do at once and slowing down all of the operations (maybe due to too many threads or using virtual memory). This is still an important unanswered question.

A third problem was that the multiprocessing and mpi4py modules cannot be used together. The author of mpi4py mentioned that any shared memory module that uses the os.fork function will break mpi4py, and I could not find a shared memory module that did not use os.fork. Multiprocessing seemed like it was different than other shared memory modules because it used multiple processors rather than multiple threads (see handythread), but it still broke mpi4py. Specifically, mpi4py's barrier command appeared to hang for some processors, meaning some processors would never get past a barrier. Similar effects were seen for blocking communications that contained a barrier, such as gather and reduce. I (Brandt) briefly tried to write my own MPI implementation that simply wrote pickle files and mimicked the behavior of mpi4py, but had strange problems (some processors didn't see files that appeared to exist) and gave up. A file-based MPI implementation would have been very inefficient between nodes; the motivation was that there would be very little communication between nodes and a slow MPI implementation would be made up for with the advantages of shared memory.

A fourth problem was the inconvenience of the multiprocessing module interface. Specifically, the Pool.map() function took a function and a list of arguments and applies the function to each element in the list. However, if the function took multiple arguments, Pool.map() couldn't be directly used. We wrote a function to map any function to a new function that accepted only one argument (a tuple which was unpacked). It can be found in old shared memory revisions in the util module, "eval_func_tuple". This function solved this problem. A related problem was that Pool.map() could also only evaluate functions that were directly callable from imported modules, or something similar to this. This meant that member functions in classes, which are not directly callable (there must be an instance of the class), could not be called within a Pool.map(). We managed to circumvent this restriction for our purposes as well, but there were often frustrating bugs. Further, we had to place more restrictions on the user-defined functions and there often confusing error messages.

For all of these reasons, I do not recommend attempting to implement a hybrid parallelization (shared memory intranode and distributed memory MPI internode) until either the multiprocessing module is improved or another module can better handle our needs. The only advantage I see is that the send/receive pairs can be omitted for processors within a node. I haven't checked the scaling carefully, but I believe this will not significantly change the computational scaling of our current implementation.

6 Additional Utilities

There are some nice utility functions in the util module, util.py. For example, there are functions for saving and loading 1D and 2D arrays/matrices and for taking a standard inner product. These functions are tested and reliable, and should be used when possible.

7 Final notes and tips

If you save files or print to the screen in your main script while running in parallel, then you will be saving or printing multiple times. As a first try, try to do everything as if you are writing a serial program. Modaldecomp has been written so that you don't have to think about parallization. However, sometimes you will need to avoid multiple processors saving to the same file or printing to the screen at the same time. You can solve this by putting your save and print commands inside an `if` block.

```
#At the beginning of your main script, add this
import parallel as parallel_mod
parallel = parallel_mod.parallelInstance

# Then to print or save, use this if block
if parallel.isRankZero():
    print 'hi'
```

The code above will always work, even if mpi4py is not installed and/or you are running serial.

It is possible to use modaldecomp in ways it wasn't intended for. For example, you could have all of your data saved to one file and load_field function could be passed a "path" that is an integer, rather than a string, which corresponds to a time-step. You might get away with this if you're lucky, you might not, and if you do, a later version might break how you use it. We don't guarantee this to work. To be safe, follow the usage suggested in this document and the sphinx documentation. It is worth noting that in the future we may try to rework load and save to be generic input and output functions so that it is not necessary to always save and load from file.

References

[1] C. W. Rowley. Model reduction for fluids using balanced proper orthogonal decomposition. *Int. J. Bifurcation Chaos*, 15(3):997–1013, March 2005.