Using PyNIO and MPI for Python to help solve a big data problem for climate modelers

David Brown (dbrown@ucar.edu)
John Dennis
Kevin Paul
Sheri Mickelson
Wei Huang
Outline

• What is the problem?
• The “time slice to time series bake-off”
• The surprisingly simple Python-based PyNIO/MPI code that prevailed
• PyNIO and how it compares to other scientific data I/O packages
• Where do we go from here?
Climate model data in a nutshell

- **100s of variables define model state**
  - Propelled by equations modeling the physics
  - Each timestep: the instantaneous state of the model
  - 2d for single layers (e.g. 1° grid: 180 lat x 360 lon)
  - 3d for multiple levels (e.g. 50 lev x 180 lat x 360 lon)

- **Multiple model components**
  - Atmosphere, ocean, land, ice, bio-geochemistry, etc.
  - Data periodically exchanged across boundaries of components: “Coupled model”
  - Output volume varies by component and grid resolution

- **Model time covers 100s – 1000s of years**
  - Raw output is all variables at a single time step: a “history file”
  - Usually consolidated into temporal averages for manageability
### Climate Data Volume example

<table>
<thead>
<tr>
<th></th>
<th>CMIP3</th>
<th>CMIP5</th>
<th>CMIP6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of participating models</strong></td>
<td>21</td>
<td>45</td>
<td>?</td>
</tr>
<tr>
<td><strong>IPCC report</strong></td>
<td>AR4</td>
<td>AR5</td>
<td>AR6</td>
</tr>
<tr>
<td><strong>NCAR community model</strong></td>
<td>CCSM3</td>
<td>CESM1</td>
<td>CESM?</td>
</tr>
<tr>
<td><strong>Model Output</strong></td>
<td>120 TB</td>
<td>1.2 PB</td>
<td>?</td>
</tr>
<tr>
<td><strong>Converted to time series</strong></td>
<td>n/a</td>
<td>760 TB</td>
<td>?</td>
</tr>
<tr>
<td><strong>Provided to CMIP archive</strong></td>
<td>10 TB</td>
<td>170 TB</td>
<td>?</td>
</tr>
</tbody>
</table>

**IPCC** : Intergovernmental Panel on Climate Change  
**CMIP** : Coupled Model Intercomparison Project  
**CCSM** : Community Climate System Model  
**CESM** : Community Earth System Model  
**AR** : Assessment Report
What scientists want from the data

• Concentrate on a few key variables over a period of time

• Raw model output is inconvenient:
  – Need to stitch together data from many files to get a time series
  – A lot of space taken by the uninteresting variables

• Solution: transpose the data into files that contain one variable with many time steps.
Time-Slice to Time-Series Conversion

Slice 5  Slice 4  Slice 3  Slice 2  Slice 1
Field 2  Field 2  Field 2  Field 2  Field 1
Field 2  Field 2  Field 2  Field 2  Field 1
Field 3  Field 3  Field 3  Field 3  Field 3
Field 3  Field 3  Field 3  Field 3  Field 3
Field 3  Field 3  Field 3  Field 3  Field 3
Field 3  Field 3  Field 3  Field 3  Field 3

Series 1
Field 1

Series 2
Field 2

Series 3
Field 3
Time slice to time series bakeoff

- Experimental bake-off with candidate tools
  - Test data suite: 10 years worth of history file output
  - Includes the typical components at various resolutions
  - Using yellowstone at NCAR/Wyoming Supercomputer Center

- 6 tools in initial experiment:
  - NCO: NetCDF Operators, the current tool (serial I/O)
  - CDO: Climate Data Operators (serial I/O)
  - NCL: NCAR Command Language (serial I/O)
  - Pagoda: NCO work-alike with data-parallel capabilities, primarily designed for working with geodesic grids
  - ncReshaper: prototype data parallel tool: Fortran and PIO
  - pyReshaper: prototype constructed using mpi4py and PyNIO
Pagoda
ncReshaper (Fortran + PIO)

pyReshaper (mpi4py + PyNIO)
# Current Time Series Generation Tool

<table>
<thead>
<tr>
<th>Datasets – 10 yrs monthly history files</th>
<th>Size (Gbytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAMFV-1.0</td>
<td>28</td>
</tr>
<tr>
<td>CAMSE-1.0</td>
<td>31</td>
</tr>
<tr>
<td>CICE-1.0</td>
<td>8</td>
</tr>
<tr>
<td>CAMSE-0.25</td>
<td>1077</td>
</tr>
<tr>
<td>CLM-1.0</td>
<td>9</td>
</tr>
<tr>
<td>CLM-0.25</td>
<td>84</td>
</tr>
<tr>
<td>CICE-0.1</td>
<td>570</td>
</tr>
<tr>
<td>POP-0.1</td>
<td>3184</td>
</tr>
<tr>
<td>POP-1.0</td>
<td>194</td>
</tr>
</tbody>
</table>

*** Comparing the time it takes to convert 10 years of monthly time slice data to time series data using the existing method

- NCO version w/coord variables: 14 hours
- 5 hours
Initial comparison: throughput rates

Throughput for slice-2-series conversion w/coord variables

- NCO
- CDO
- NCL 6.1.2
- Pagoda
- ncReshaper (Fortran + PIO)
- pyReshaper (pyNIO + mpi4py)

datasets
Duration

<table>
<thead>
<tr>
<th>Dataset</th>
<th>NCO</th>
<th>ncReshaper</th>
<th>pyReshaper</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>POP – 0.10</td>
<td>14 ½ hours</td>
<td>2 ½ hours</td>
<td>1 ½ hours</td>
<td>~ 8 X</td>
</tr>
<tr>
<td>CAMSE – 0.25</td>
<td>4 ¾ hours</td>
<td>2 hours</td>
<td>28 minutes</td>
<td>~ 10 X</td>
</tr>
</tbody>
</table>
Throughput Rates

Throughput for slice-2-series conversion w/coord variables

- NCO
- ncReshaper (Fortran + PIO)
- pyReshaper (pyNIO + mpi4py)

MB/second vs. datasets
pyReshaper features

- Input files shared among tasks
- One task per variable
- One task per output file
- Simple MPI code
  - no critical sections
  - no syncing required
- Admittedly designed for a single specialized task
- But already has provided a template for a new temporal averaging tool
class MPIMessenger(Messenger):

def __init__(self):
    from mpi4py import MPI
    self._mpi = MPI
    self._mpi_rank = self._mpi.COMM_WORLD.Get_rank() # identifies this process
    self._mpi_size = self._mpi.COMM_WORLD.Get_size() # total number of processes
    self._is_master = (self._mpi_rank == 0) # for convenience

    # this method is the only one needed to divide the work up among the processes
    def partition(self, global_list):
        local_list = global_list[self._mpi_rank::self._mpi_size]
        return local_list

    # other Messenger class methods needed only to keep track of execution statistics
pyReshaper contains:

class Messenger -- interfaces with MPI
class Reshaper -- performs the actual work
class Specifier -- specifies input files and output options
class TimeKeeper -- timers used to gather data on phases of the execution

class MPIMessenger(Messenger):

    def __init__(self):

        from mpi4py import MPI
        self._mpi = MPI  # Pointer to the MPI module
        self._mpi_rank = self._mpi.COMM_WORLD.Get_rank()  # identifies this process
        self._mpi_size = self._mpi.COMM_WORLD.Get_size()  # total number of processes
        self._is_master = (self._mpi_rank == 0)  # for convenience

        # this method is the only one needed to divide the work up among the processes

    def partition(self, global_list):

        local_list = global_list[self._mpi_rank::self._mpi_size]
        return local_list

    # other Messenger class methods needed only to keep track of execution statistics
class Reshaper(object):
    def __init__(self, spec, serial=False, verbosity=1):
        self._messenger = create_messenger(serial=serial)
        self._specifier = spec

        for filename in self._specifier.input_file_list:
            self._input_files.append(Nio.open_file(filename, "r"))

        self._validate_and_sort_input_files()
        self._sort_and_classify_variables()  # data or metadata (time-invariant or varying)

    def convert(self):
        # simplified
        local_ts_varnames = self._messenger.partition(self._time_series_variables)

        for out_name in local_ts_varnames:
            out_filename = self._specifier.get_output_filename(out_name)
            out_file = Nio.open_file(out_filename, 'w', options=self._nio_options)

            in_var = self._input_files[0].variables[out_name]
            out_var = out_file.create_variable(out_name, in_var.typecode(), in_var.dimensions)

            t_index = 0
            for in_file in self._input_files:
                # note time dimension unlimited: grows by one each iteration
                out_var[t_index] = in_file.variables[out_name][:]
                t_index += 1

            out_file.close()
class Reshaper(object):
    def __init__(self, spec, serial=False, verbosity=1):
        self._messenger = create_messenger(serial=serial)
        self._specifier = specifier

        for filename in self._specifier.input_file_list:
            self._input_files.append(Nio.open_file(filename, "r"))

        self._validate_and_sort_input_files()
        self._sort_and_classify_variables()  # data or metadata (time-invariant or varying)

    def convert(self):
        # simplified
        local_ts_varnames = self._messenger.partition(self._time_series_variables)

        for out_name in local_ts_varnames:
            out_filename = self._specifier.get_output_filename(out_name)
            out_file = Nio.open_file(out_filename, 'w', options=self._nio_options)

            in_var = self._input_files[0].variables[out_name]
            out_var = out_file.create_variable(out_name, in_var.typecode(), in_var.dimensions)

            t_index = 0
            for in_file in self._input_files:
                # note time dimension unlimited: grows by one each iteration
                out_var[t_index] = in_file.variables[out_name][::]
                t_index += 1

            out_file.close()
class Reshaper(object):
    def __init__(self, spec, serial=False, verbosity=1):
        self._messenger = create_messenger(serial=serial)
        self._specifier = specifier
        for filename in self._specifier.input_file_list:
            self._input_files.append(Nio.open_file(filename, "r"))

        self._validate_and_sort_input_files()
        self._sort_and_classify_variables()  # data or metadata (time-invariant or varying)

    def convert(self):
        # simplified
        local_ts_varnames = self._messenger.partition(self._time_series_variables)

        for out_name in local_ts_varnames:
            out_filename = self._specifier.get_output_filename(out_name)
            out_file = Nio.open_file(out_filename, 'w', options=self._nio_options)

            in_var = self._input_files[0].variables[out_name]
            out_var = out_file.create_variable(out_name, in_var.typecode(), in_var.dimensions)

            t_index = 0
            for in_file in self._input_files:
                # note time dimension unlimited: grows by one each iteration
                out_var[t_index] = in_file.variables[out_name][:]
                t_index += 1

            out_file.close()
class Reshaper(object):
    def __init__(self, spec, serial=False, verbosity=1):
        self._messenger = create_messenger(serial=serial)
        self._specifier = spec

        for filename in self._specifier.input_file_list:
            self._input_files.append(Nio.open_file(filename, "r"))

        self._validate_and_sort_input_files()
        self._sort_and_classify_variables()  # data or metadata (time-invariant or varying)

    def convert(self):
        # simplified
        local_ts_varnames = self._messenger.partition(self._time_series_variables)

        for out_name in local_ts_varnames:
            out_filename = self._specifier.get_output_filename(out_name)
            out_file = Nio.open_file(out_filename, 'w', options=self._nio_options)

            in_var = self._input_files[0].variables[out_name]
            out_var = out_file.create_variable(out_name, in_var.typecode(), in_var.dimensions)

            t_index = 0
            for in_file in self._input_files:
                # note time dimension unlimited: grows by one each iteration
                out_var[t_index] = in_file.variables[out_name][:]
                t_index += 1

        out_file.close()
class Reshaper(object):
    def __init__(self, spec, serial=False, verbosity=1):

        self._messenger = create_messenger(serial=serial)
        self._specifier = specifier

        for filename in self._specifier.input_file_list:
            self._input_files.append(Nio.open_file(filename, "r"))

        self._validate_and_sort_input_files()
        self._sort_and_classify_variables()  # data or metadata (time-invariant or varying)

    def convert(self):
        # simplified
        local_ts_varnames = self._messenger.partition(self._time_series_variables)

        for out_name in local_ts_varnames:
            out_filename = self._specifier.get_output_filename(out_name)
            out_file = Nio.open_file(out_filename, 'w', options=self._nio_options)

            in_var = self._input_files[0].variables[out_name]
            out_var = out_file.create_variable(out_name, in_var.typecode(), in_var.dimensions)

            t_index = 0
            for in_file in self._input_files:
                # note time dimension unlimited: grows by one each iteration
                out_var[t_index] = in_file.variables[out_name][:]
                t_index += 1

            out_file.close()
class Reshaper(object):
    def __init__(self, spec, serial=False, verbosity=1):
        self._messenger = create_messenger(serial=serial)
        self._specifier = specifier

        for filename in self._specifier.input_file_list:
            self._input_files.append(Nio.open_file(filename, "r"))

        self._validate_and_sort_input_files()
        self._sort_and_classify_variables()  # data or metadata (time-invariant or varying)

    def convert(self):
        # simplified
        local_ts_varnames = self._messenger.partition(self._time_series_variables)

        for out_name in local_ts_varnames:
            out_filename = self._specifier.get_output_filename(out_name)
            out_file = Nio.open_file(out_filename, 'w', options=self._nio_options)

            in_var = self._input_files[0].variables[out_name]
            out_var = out_file.create_variable(out_name, in_var.typecode(), in_var.dimensions)

            t_index = 0
            for in_file in self._input_files:
                # note time dimension unlimited: grows by one each iteration
                out_var[t_index] = in_file.variables[out_name][:]
                t_index += 1

            out_file.close()
class Reshaper(object):
    def __init__(self, spec, serial=False, verbosity=1):
        self._messenger = create_messenger(serial=serial)
        self._specifier = specifier

        for filename in self._specifier.input_file_list:
            self._input_files.append(Nio.open_file(filename, "r"))

        self._validate_and_sort_input_files()
        self._sort_and_classify_variables()  # data or metadata (time-invariant or varying)

    def convert(self):
        # simplified
        local_ts_varnames = self._messenger.partition(self._time_series_variables)

        for out_name in local_ts_varnames:
            out_filename = self._specifier.get_output_filename(out_name)
            out_file = Nio.open_file(out_filename, 'w', options=self._nio_options)

            in_var = self._input_files[0].variables[out_name]
            out_var = out_file.create_variable(out_name, in_var.typecode(), in_var.dimensions)

            t_index = 0
            for in_file in self._input_files:
                # note time dimension unlimited: grows by one each iteration
                out_var[t_index] = in_file.variables[out_name][:]
                t_index += 1

            out_file.close()
mpi4py makes it work

• PyNIO is serial code
• Like NetCDF4Python derived from Konrad Hinson’s Scientific NetCDF module
• Simple to substitute NetCDF4Python
• So how would it compare?
  – NetCDF3
  – NetCDF4 classic
  – NetCDF4 classic with compression level 1
Why PyNIO?

- NetCDF-style interface to formats of interest to the atmospheric science community
  - NetCDF3, NetCDF4 classic, with full NetCDF4 coming soon
  - HDF4, HDF5, HDF-EOS2, HDF-EOS5
  - GRIB1 and GRIB2
  - Shapefiles
- Well-supported: 20+ years of development
- Flexible interface to MaskedArrays
  - Full support for _FillValue and missing_value
- Added value coordinate arrays
  - Geographical (GRIB and HDFEOS)
  - Vector rotations (GRIB)
  - Time (GRIB)
Next PyNIO version

• Support for full NetCDF4 interface
  – Groups
  – Vlen (ragged arrays) and compound data
  – Strings and all integer types
• Easy install using conda
• Many format-specific improvements
• Improved compatibility with other tools
• Version 1.5.0 to be released soon
A modular tool suite

• Derived from NCL
• Import only what you need
• Easily cooperate with other Python tools
• PyNIO for file I/O
• PyNGL for graphics
• New Python analysis library
  – Will contain the significant NCL functions
  – Complementary to Aoslib
• Commitment to Python community
Thanks

• John Dennis for original pyNIO/mpi4py code and initiating the bake off
• Sheri Mickelson for generating test results
• Kevin Paul for productizing pyReshaper
• Wei Huang (huangwei@ucar.edu) for adding NetCDF4 capabilities to PyNIO
• The mpi4py developers and the Python community for creating great infrastructure

dbrown@ucar.edu
Throughput for slice-2-series conversion w/coord variables

- NCO
- NCL 6.1.2
- pyNIO
- pyNIO + MPI
- ncReshaper
- Pagoda
- CDO

MBytes/sec vs datasets

CAMSE-1.0
Other methods that were tested

Duration

Throughput

![Duration slice-2-series conversion w/coord variables](image1)

![Throughput for slice-2-series conversion w/coord variables](image2)
Throughput
pyReshaper vs the other tools

Throughput for slice-2 series conversion w/coord variables

- NCO
- pyReshaper (NetCDF3)
- pyReshaper (NetCDF4)
- pyReshaper (NetCDF4 - Compression Level 1)

Throughput for slice-2-series conversion w/coord variables

- NCO
- NCL 6.1.2
- Pagoda
- CDO
PyReshaper testing results for different netCDF types

Duration

Throughput

Duration slice-2-series conversion w/coord variables

Throughput for slice-2-series conversion w/coord variables
pyNIO vs netCDF4Python max
Climate data volume

• CCSM3 data for CMIP3 (~2004-2006):
  – Input to IPCC AR4 (one of 21 participating models)
  – 120 TB history files
  – 10 TB provided to the CMIP3 archive

• CESM data for CMIP5 (-> July 2012):
  – Input to IPCC AR5 (one of 45 participating models)
  – 1.2 PB data as history files (time slice)
  – 760 TB converted to time series by the deadline
  – 170 TB provided to CMIP5 archive

• Projected to grow exponentially for CMIP6
  – Of course!
Time Series Generation Tool

History Time Slice Files

<table>
<thead>
<tr>
<th>Time 1</th>
<th>Time m</th>
</tr>
</thead>
<tbody>
<tr>
<td>header</td>
<td>header</td>
</tr>
<tr>
<td>Field 1</td>
<td>Field 1</td>
</tr>
<tr>
<td>Field 2</td>
<td>Field 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Field n</td>
<td>Field n</td>
</tr>
</tbody>
</table>

Transposed to Time Series Files

<table>
<thead>
<tr>
<th>Field n+1</th>
<th>Header</th>
<th>Time 1</th>
<th>Time 2</th>
<th>...</th>
<th>Time m</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field n+2</td>
<td>Header</td>
<td>Time 1</td>
<td>Time 2</td>
<td>...</td>
<td>Time m</td>
</tr>
<tr>
<td>Field n+x</td>
<td>Header</td>
<td>Time 1</td>
<td>Time 2</td>
<td>...</td>
<td>Time m</td>
</tr>
</tbody>
</table>
So what is the problem?

- Huge and ever increasing data volume
- Raw output in the form of “history files”
  - i.e. the values of all variables at each output time step.
- Not a convenient arrangement of the data for a research scientist.
- Converting the data into a usable form has become a serious bottleneck.
class MPIMessenger(Messenger):

    def __init__(self):
        from mpi4py import MPI
        self._mpi = MPI  # Pointer to the MPI module
        self._mpi_rank = self._mpi.COMM_WORLD.Get_rank()  # identifies this process
        self._mpi_size = self._mpi.COMM_WORLD.Get_size()  # total number of processes
        self._is_master = (self._mpi_rank == 0)  # for convenience

        # this method is the only one needed to divide the work up among the processes

    def partition(self, global_list):
        local_list = global_list[self._mpi_rank::self._mpi_size]
        return local_list

    # other Messenger class methods needed only to keep track of execution statistics

pyReshaper contains:

    class Messenger   --  interfaces with MPI, top-level class is serial
    class Reshaper     --  performs the actual work
    class Specifier    --  specifies input files and output options
    class TimeKeeper   --  timers used to gather data on phases of the execution