

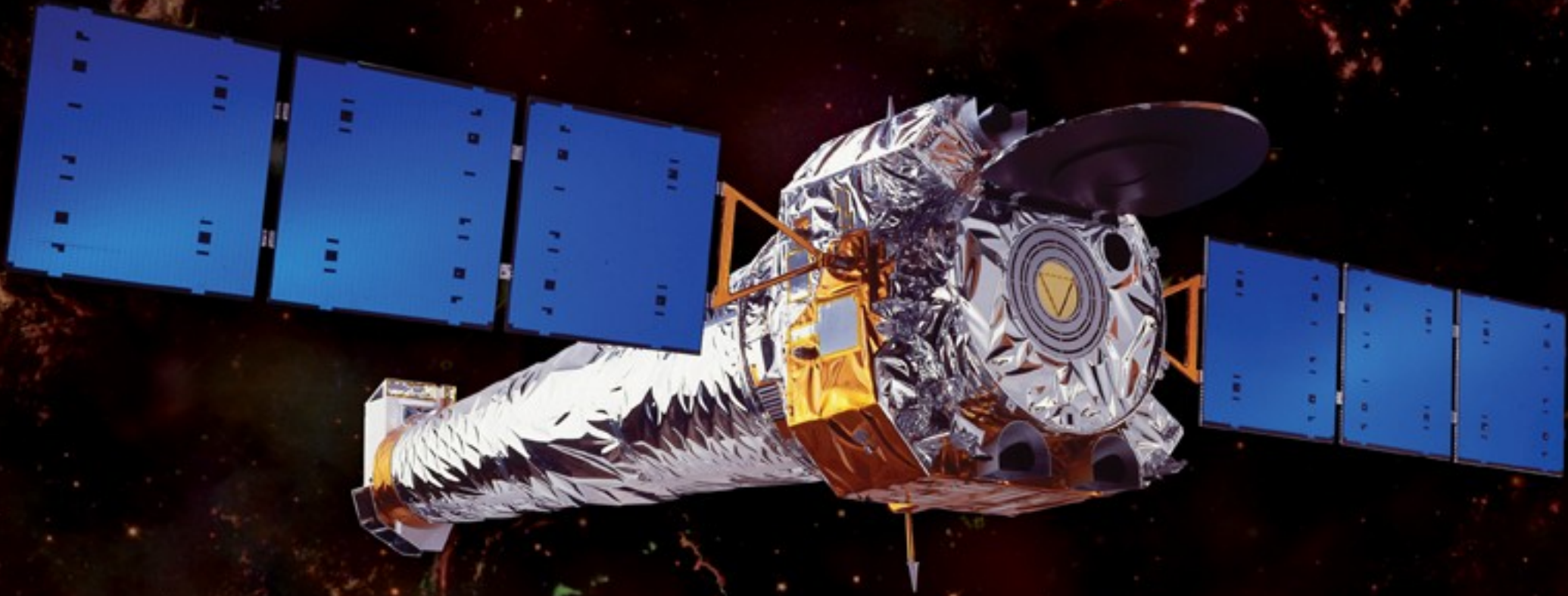


Keeping the Chandra Satellite Cool with Python

**Tom Aldcroft
CXC Operations Science Support
Smithsonian Astrophysical Observatory**

**SciPy2010 conference
July 1, 2010**

Chandra satellite (at launch)



- Chandra X-ray Observatory launched by NASA in July 1999
- X-rays reveal energetic processes from black holes, supernova explosions, massive galaxy clusters, pulsars, and more
- Factor of 8 better angular resolution than any other X-ray observatory
- Operating superbly since launch with many ground-breaking discoveries

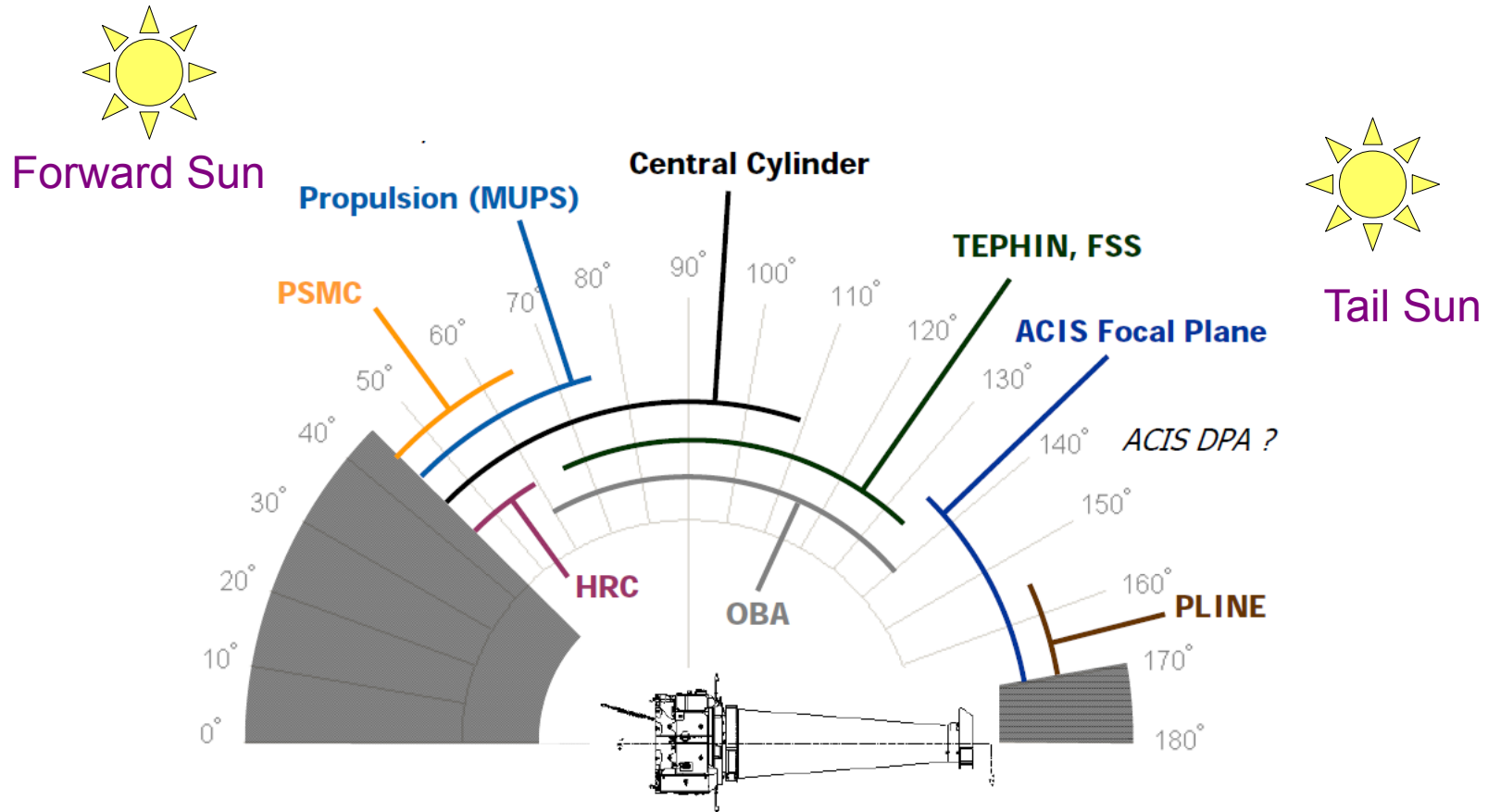
Chandra satellite (now)



- Ionizing particle radiation environment was worse than expected and was degrading the silverized teflon insulation
- Early in the mission spacecraft temperatures began rising faster than expected
- No repair possible



Impact to operations



- Each year observers submit proposals, selected on scientific merit
- Need to get to each target (nightmare traveling salesman problem)
- Until recently used very simple thermal models and constraints to maintain spacecraft components within thermal limits
- *Major driver in operations and mission scheduling*

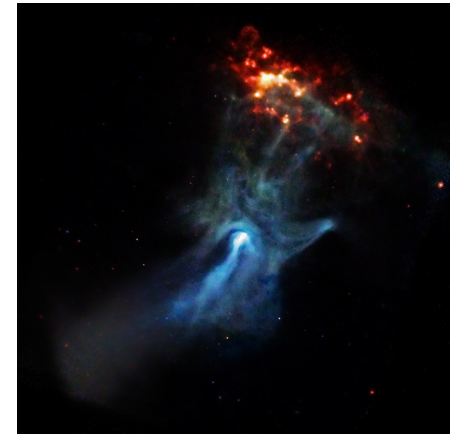


Thermal modeling

- Eventually the simple models became inadequate
- In 2007 a joint working group including scientists and engineers was formed to address thermal issues and develop higher fidelity thermal models.

Python was chosen to support this effort:

- Ease of development and strong interactive analysis environment
- NumPy (consolidated numerical support)
- IPython
- Matplotlib
- SciPy
- Other 3rd party packages (PyTables, Sherpa, etc)



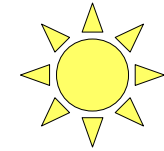


Telemetry access

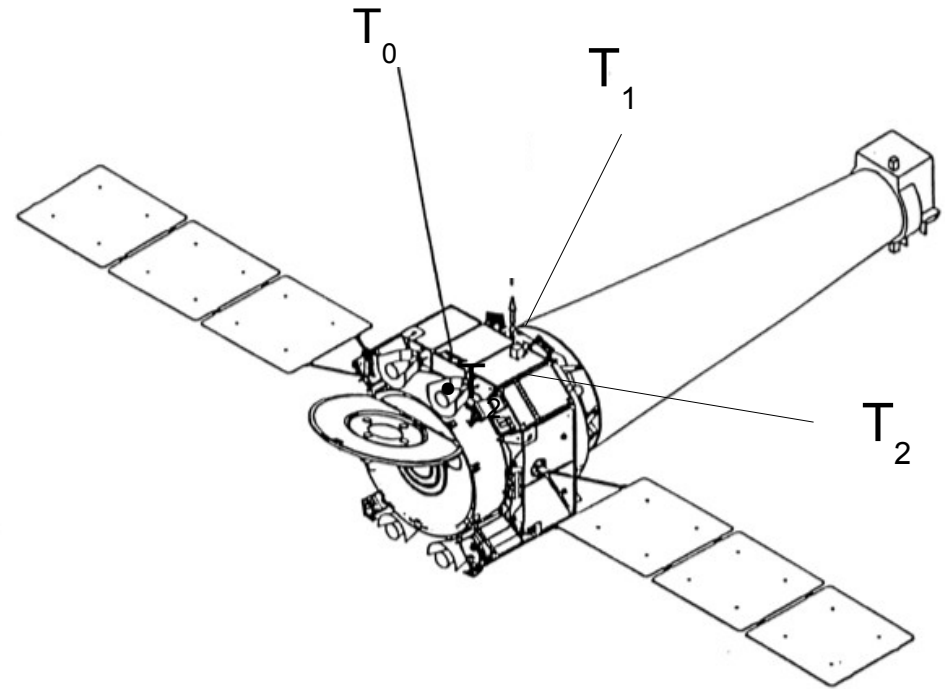
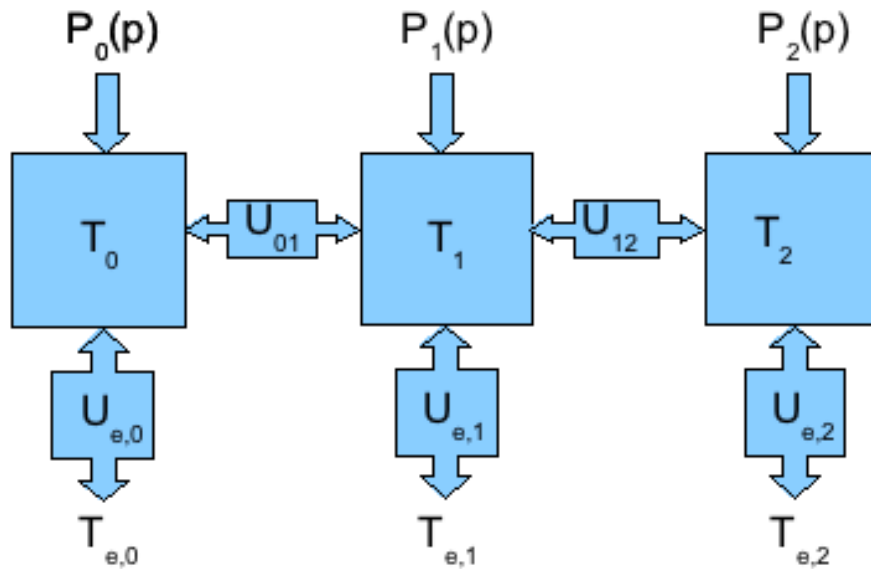
Problem: Developing models requires fast access to years of thermal data

- Chandra records science and engineering data onboard at 32 kbit/s
- Engineering data consists of over 6000 distinct data items (“MSIDs”)
- Data volume meager but telemetry access tools spec'd to 1990's hardware
 - These tools can take hours to retrieve a year of data (10^7 elements/hour...)
 - One tool always decommutates from raw telemetry
 - Other method puts hundreds of related MSIDs in a file spanning just a few hours
=> getting one MSID is very inefficient
- **PyTables and HDF5 to the rescue:**
 - Tables with 10^{10} elements are no problem, fast random access and easy appends
 - Solution: *use one huge table per MSID covering the entire duration of the mission*
 - Compression especially efficient since many MSIDs change slowly
 - Entire telemetry archive is less than 300 Gb
 - PyTables is simple to use with good documentation and examples
 - Data retrieval speed $\sim 10^7$ elements/sec from NetApp disk (**fast enough**)

A data-driven thermal model



Sun



- Each box (T_0 , T_1 , T_2) represents a spacecraft node with a thermistor
- Node i has an external heat input $P_i(p)$ and conductances $U_{i,j}$ to other nodes
- Throw in an ad-hoc external heat bath with temperature $T_{e,i}$
- $P_i(p)$: spacecraft pitch angle to Sun and possibly internal electronics heat



A data-driven thermal model

- This model has an exact analytic solution with a nice matrix formulation:

$$\begin{aligned}\dot{\mathbf{T}} &= \mathbf{\tilde{A}}\mathbf{T} + \mathbf{b} \\ \mathbf{T}(t) &= \int_0^t e^{\mathbf{\tilde{A}}(t-u)} \mathbf{b} du + e^{\mathbf{\tilde{A}}t} \mathbf{T}(0) \\ &= [\mathbf{v}_1 \ \mathbf{v}_2] \begin{bmatrix} \frac{e^{\lambda_1 t} - 1}{\lambda_1} & 0 \\ 0 & \frac{e^{\lambda_2 t} - 1}{\lambda_2} \end{bmatrix} [\mathbf{v}_1 \ \mathbf{v}_2]^{-1} \mathbf{b} \\ &\quad + [\mathbf{v}_1 \ \mathbf{v}_2] \begin{bmatrix} e^{\lambda_1 t} & 0 \\ 0 & e^{\lambda_2 t} \end{bmatrix} [\mathbf{v}_1 \ \mathbf{v}_2]^{-1} \mathbf{T}(0)\end{aligned}$$

For a 2-node model

- The \mathbf{A} matrix depends on the conductances and \mathbf{b} vector on external power
- Accuracy is independent of step size



A fast thermal model

First implementation was a literal transcription for each time step – SLOW

Optimization steps:

- Include time dimension in arrays to compute with a single NumPy expression (factor of 10-20 improvement)
- Refactor equations to avoid any repetitions within loops
- Cache various intermediate results
- Overall factor ~40 improvement over initial literal implementation

These straightforward steps made “plain Python” code pretty fast:

Can compute a year of temperatures in ~1 sec



Fitting model parameters

These thermal models contain 15 to 80 free parameters (conductances, solar heating vs. pitch, long-term and annual variations)

CIAO Sherpa modeling and fitting package to the rescue:

- Powerful model language
 - Complex models as a Python expression
 - Parameter linking, freezing
- Good optimization methods for many-parameter fits
- Good documentation and support
- Well-suited to interactive analysis

Good reasons to worry about fitting 80 parameter models, but IT WORKS.

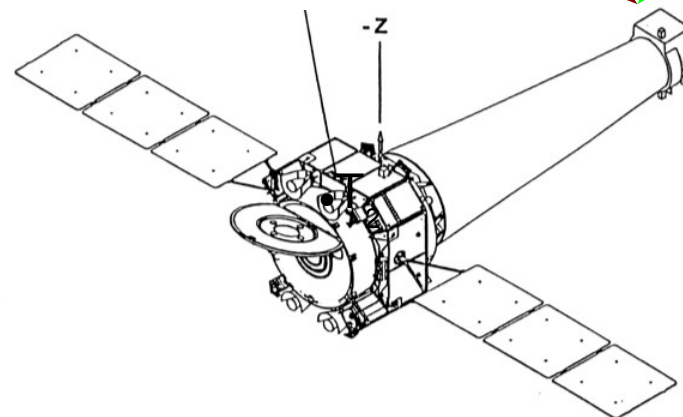
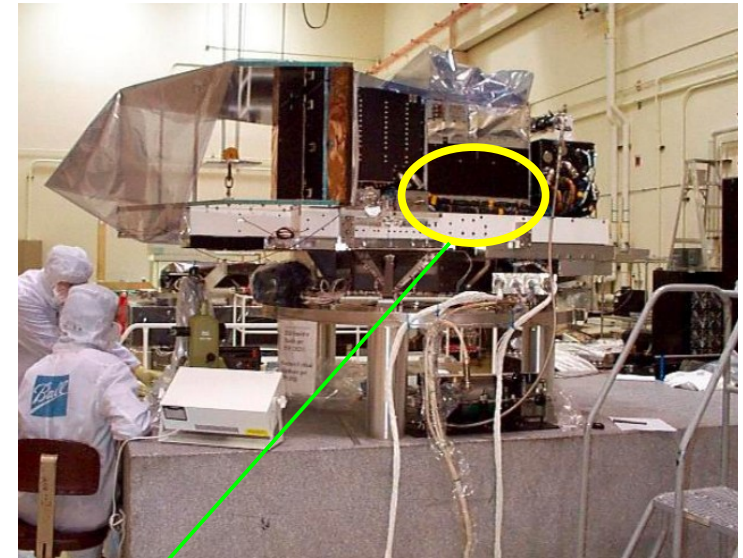
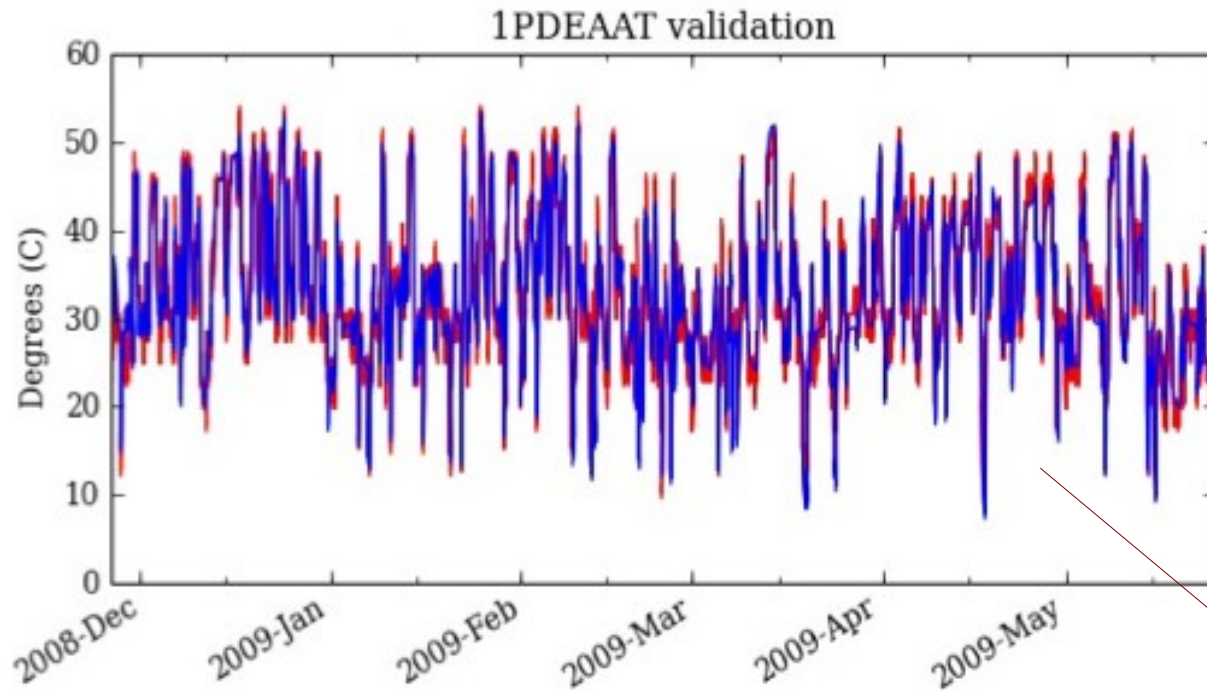


Sherpa



Calibrated model

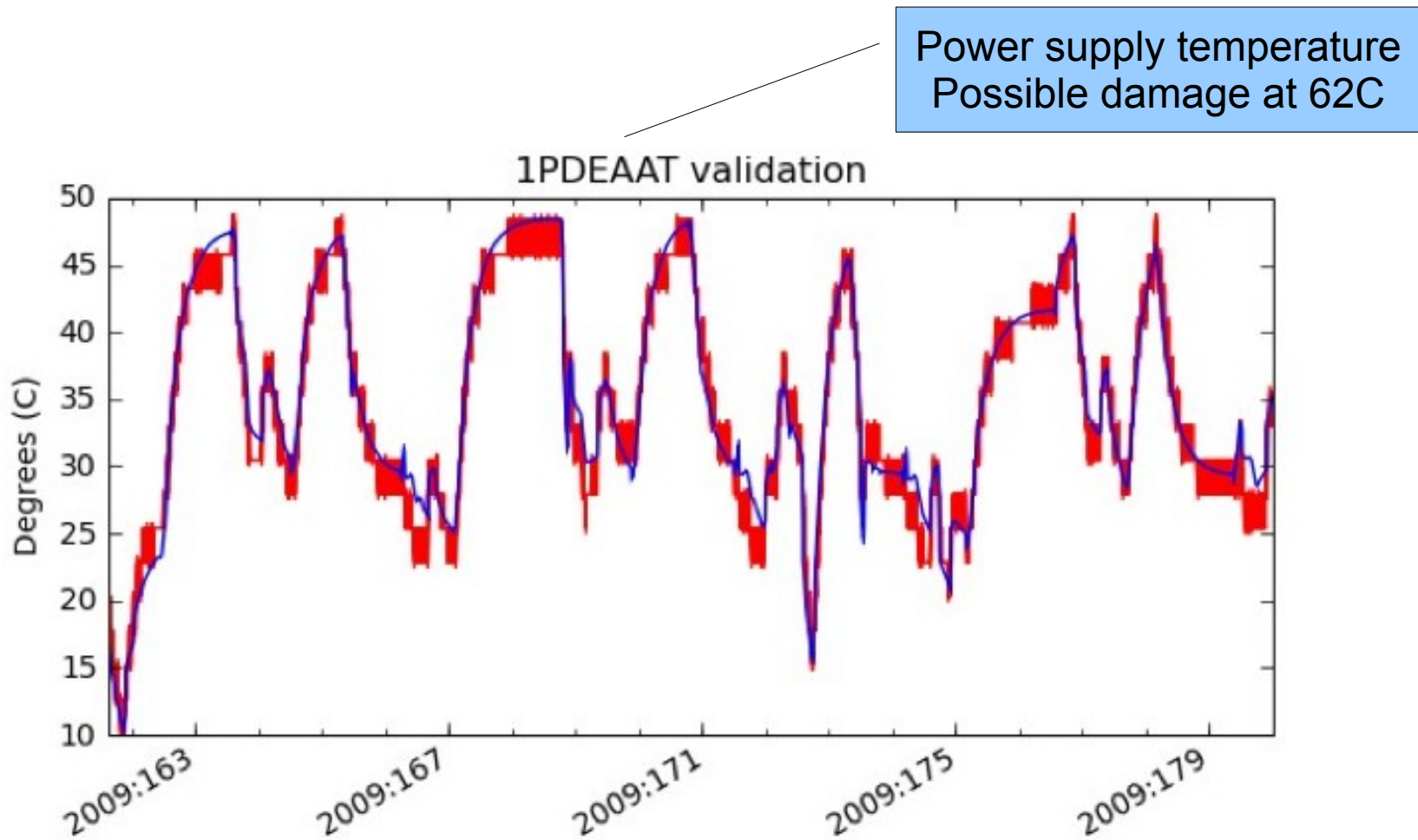
Fitting process finds parameters so the model calculation matches thermal data over the previous ~5 years (red is data, blue is model):





Calibrated model prediction

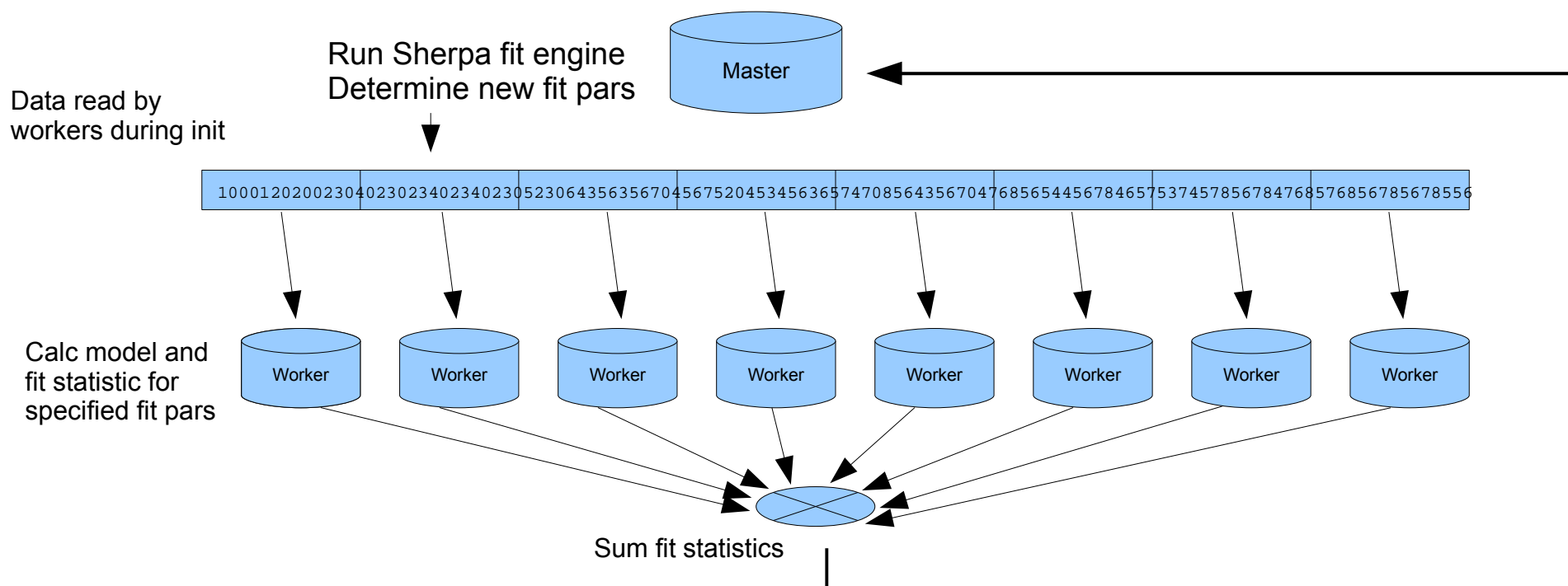
Calibrated model is then used to predict temperatures for *planned* observations





Parallelization

- Computation is easily parallelized by splitting into independent time segments
- Simple code extension with mpi4py with MPICH2
- Master-worker architecture:
 - Master controls the fit process and initialization
 - Workers read in thermal data, calculate model and χ^2 fit statistic over time segment





Parallelization with MPI

In the master program replace data initialization, model and fit statistic calculation functions with new functions:

```
comm = MPI.COMM_SELF.Spawn(sys.executable,
                            args=['fit_worker.py'],
                            maxprocs=n_workers)

def init_workers(metadata)
    """Init workers using values in metadata dict"""
    msg = {'cmd': 'init', 'metadata': metadata}
    comm.bcast(msg, root=MPI.ROOT)

def calc_model(pars):
    """Calculate the model for given pars"""
    comm.bcast(msg={'cmd': 'calc_model', 'pars': pars},
               root=MPI.ROOT)

def calc_stat():
    """Calculate chi^2 diff between model and data"""
    msg = {'cmd': 'calc_statistic'}
    comm.bcast(msg, root=MPI.ROOT)
    fit_stat = numpy.array(0.0, 'd')
    comm.Reduce(None, [fit_stat, MPI.DOUBLE],
                op=MPI.SUM, root=MPI.ROOT)
    return fit_stat
```



Parallelization with MPI

The main logic of the master fit program is nearly unchanged except for the addition of code to dynamically spawn workers:

```
init_workers({"start": date_start, "stop": date_stop})

# Sherpa commands to register and configure a function
# as a user model
load_user_model(calc_model, 'mpimod')
add_user_pars('mpimod', parnames)
set_model(mpimod)

# Configure the fit statistic
load_user_stat('mpistat', calc_stat)
set_stat(mpistat)

# Do the fit
fit()
```



Parallelization with MPI

The fit worker code just waits around to get instructions:

```
comm = MPI.Comm.Get_parent()
size = comm.Get_size()
rank = comm.Get_rank()

while True:
    msg = comm.bcast(None, root=0)

    if msg['cmd'] == 'stop':
        break

    elif msg['cmd'] == 'init':
        x, y = get_data(msg['metadata'], rank, size)

    elif msg['cmd'] == 'calc_model':
        model = calc_model(msg['pars'], x, y)

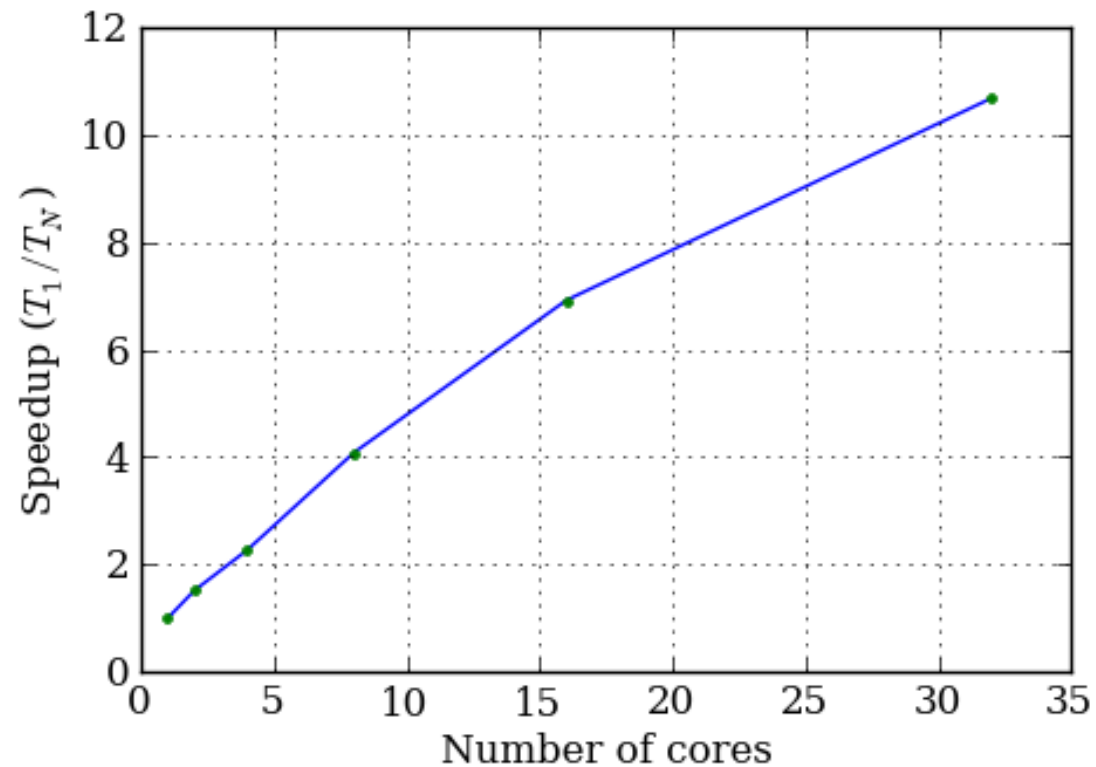
    elif msg['cmd'] == 'calc_statistic':
        fit_stat = numpy.sum((y - model)**2)
        comm.Reduce([fit_stat, MPI.DOUBLE], None,
                    op=MPI.SUM, root=0)

comm.Disconnect()
```



Parallelization speedup

- The speedup obtained is useful
- Parallel fraction = 0.94
- Ultimate speedup = 16





Putting it all together

Two certified (NASA Level-4 review board) thermal models are being used in Chandra operations:

- Power Supply and Motor Controller for the Advanced Chandra Imaging Spectrometer (ACIS PSMC)
- “Minus-Z” model of 5 nodes on the Sun-pointed (-Z) side of the spacecraft

Formal command load review process verifies no thermal limit violations in the schedule of planned observations for the next week.

Developing higher-fidelity thermal models for Chandra was a difficult thing that was made possible by the Python ecosystem.

The reason we bother

