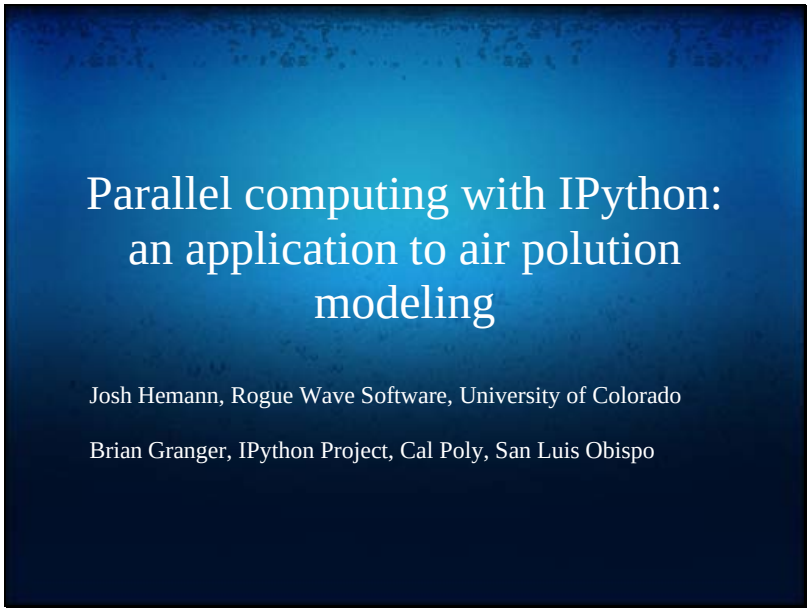
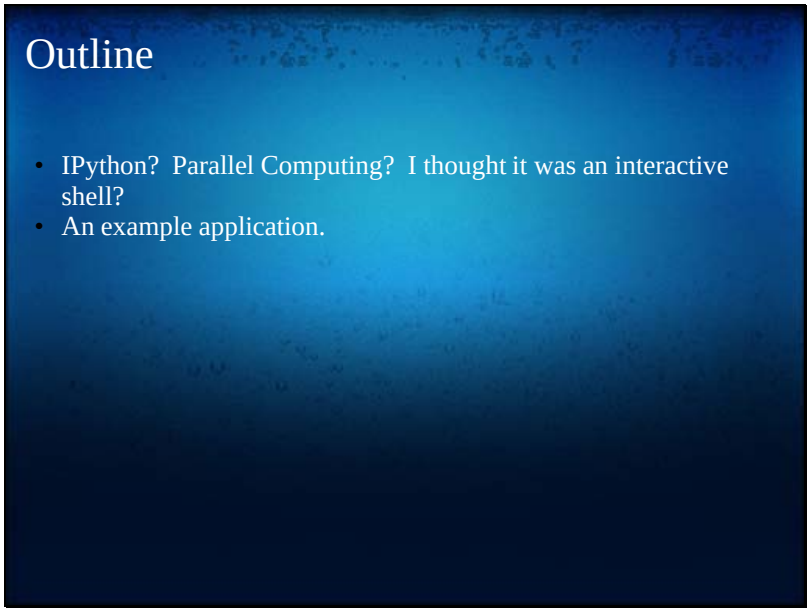
Slide 1


Parallel computing with IPython: an application to air polution modeling
Josh Hemann, Rogue Wave Software, University of Colorado
Brian Granger, IPython Project, Cal Poly, San Luis Obispo

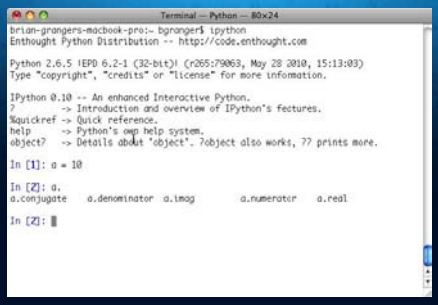
Slide 2


Outline
• IPython? Parallel Computing? I thought it was an interactive shell?
• An example application.

Slide 3

IPython Overview

- Goal: provide an efficient environment for exploratory and interactive scientific computing.
- Cross platform and open source (BSD).
- Two main components:
  - An enhanced interactive Python shell.
  - A framework for interactive parallel computing.

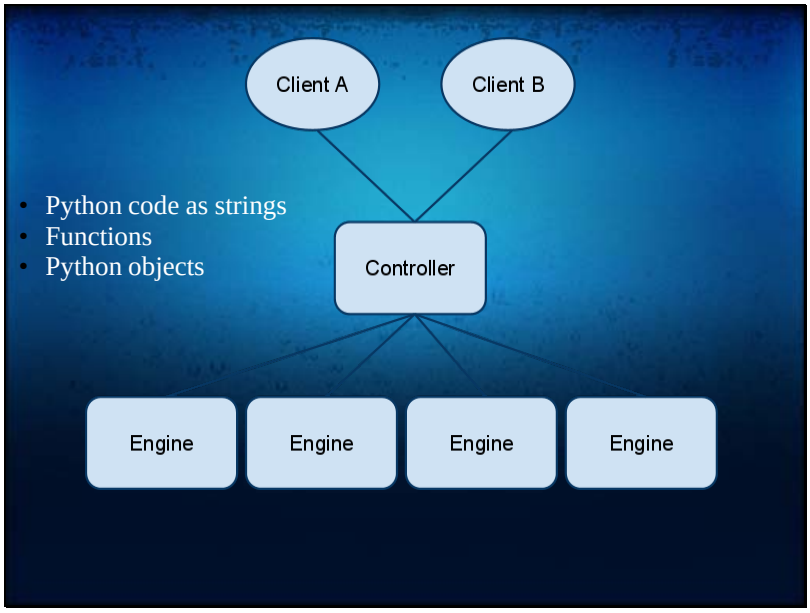


Slide 4

IPython's Parallel Framework

- Goal: provide a high level interface for executing Python code in parallel on everything: multicore CPUs, clusters, supercomputers and the cloud.
- Easy things should be easy, difficult things possible.
- Make parallel computing collaborative, interactive.
- A dynamic process model for fault tolerance and load balancing.
- Want to keep the benefit of traditional approaches:
  - Integrate with threads/MPI if desired.
  - Integrate with compiled, parallel C/C++/Fortran codes.
- Support different types of parallelism.
- Based on processes not threads (the GIL).
- Why parallel computing in IPython?
  - R(EEEEE...)PL is the same as REPL if abstracted properly.

Slide 5


Client A
Client B
Python code as strings
Functions
Python objects
Controller
Engine
Engine
Engine
Engine

Slide 6

Architecture details

- The IPython Engine is a Python interpreter that executes code received over a network.
- The Controller maintains a registry of the engines and a queue for code to be run on each engine. Handles load balancing.
- Dynamic and fault tolerant: Engines can come and go at any time.
- The Client is used in top-level code to submit tasks to the controller/engines.
- Client, Controller and Engines are fully asynchronous.
- Remote exception handling: exceptions on the engines are serialized and returned to the client.
- Everything is interactive, even on a supercomputer or the cloud.

Slide 7

MultiEngineClient and TaskClient

- MultiEngineClient
  - Provides direct, explicit access to each Engine.
  - Each Engine has an id.
  - Full integration with MPI (MPI rank == id).
  - No load balancing.
- TaskClient
  - No information about number of Engines or their identities.
  - Dynamic load balanced queue.
  - No MPI integration.
- Extensible
  - Possible to add new interfaces (Map/Reduce).
  - Not easy, but we hope to fix that.

Slide 8

Job Scheduler Support

To perform a parallel computation with IPython, you need to start 1 Controller and N Engines. IPython has an **ipcluster** command that completely automates this process. We have support for the following batch systems.

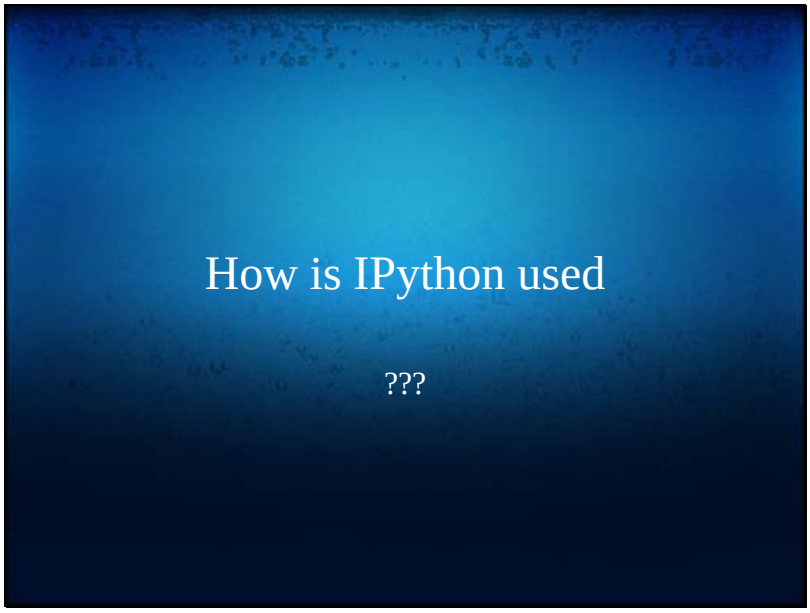
- PBS
- ssh
- mpiexe/mpirun
- SGE (coming soon)
- Microsoft HPC Server 2008 (coming soon)

Slide 9

Work in progress

- Much of our current work is being enabled by ØMQ/PyØMQ. See SciPy talk tomorrow and www.zeromq.org
- Massive refactoring of the IPython core to a two process model (frontend+kernel). This will enable the creation of long awaited GUI/Web frontends for IPython.
- Working heavily on performance and scalability of parallel computing framework.
- Simplifying the MultiEngineClient and TaskClient interfaces.

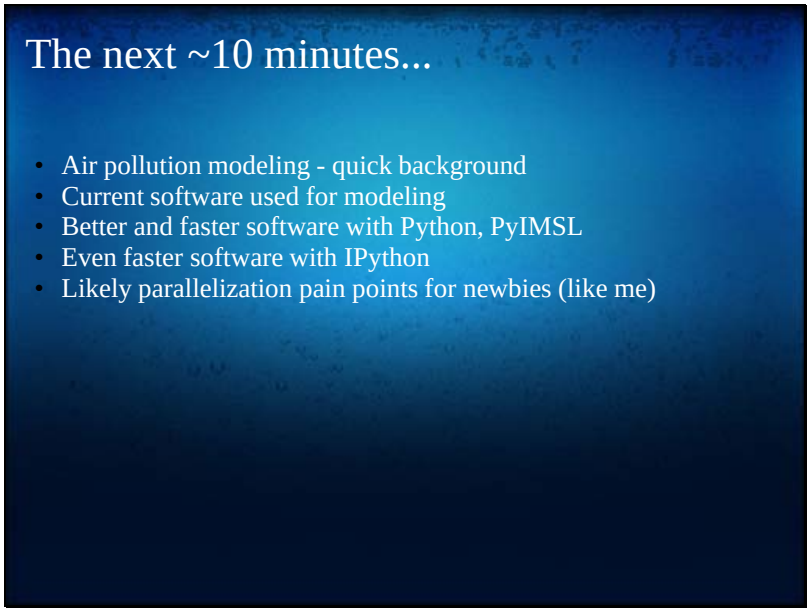
Slide 10



Slide 11

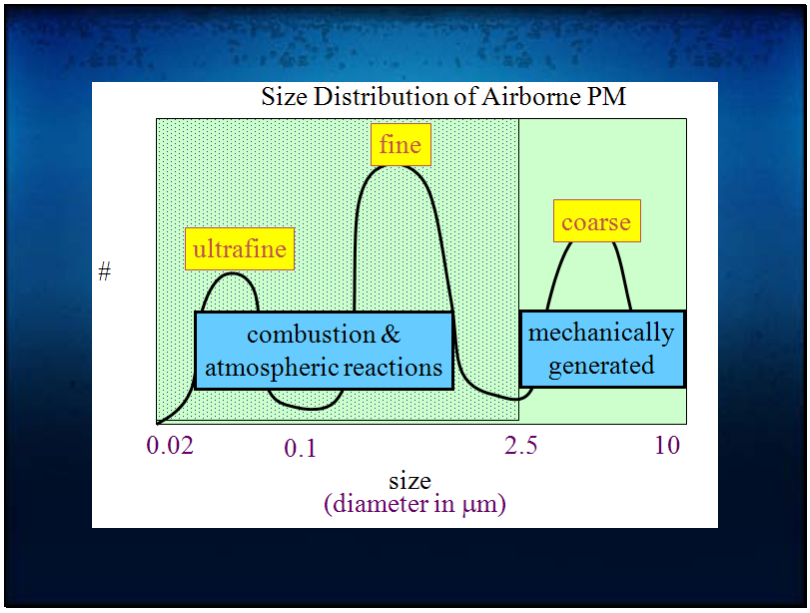
I wear two Python hats. One is for the work I do at Rogue Wave Software on our PyIMSL Studio product. I also work closely with a team of researchers (led by Dr. Mike Hannigan) doing air pollution modeling, and I use Python extensively.

Slide 12



I was at the SC show (The International Conference for High Performance Computing Networking, Storage, and Analysis) in November of 2009 when I met Wen-ming Ye, who works on Microsoft's High Performance Computing products. He put me in touch with Brian Granger, who helped me get up to speed quickly and parallelize my modeling application. This talk covers some things I have learned along the way.

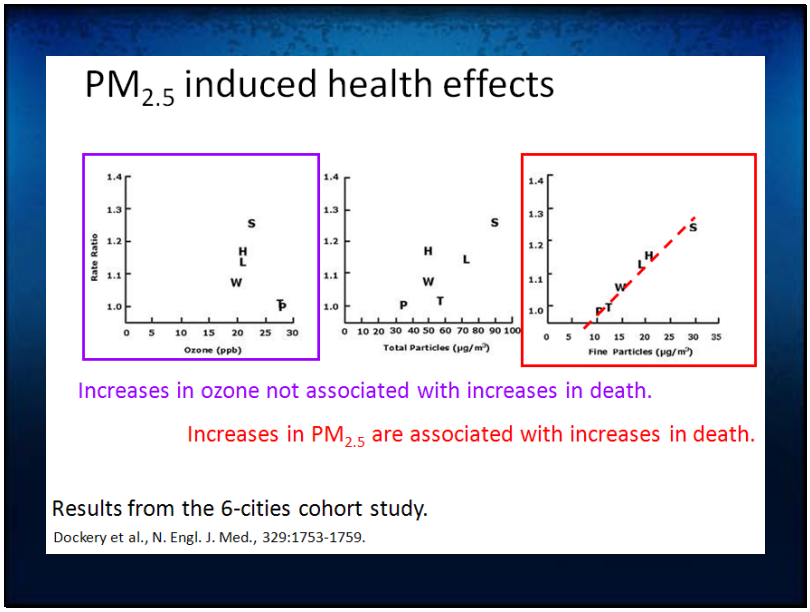
Slide 13



Since we are responsible for much of the fine and ultrafine particulate matter (through burning stuff like gasoline, coal, wood,…) , we might ask ourselves what are the effects on our health? The study of PM2.5 concerns the sources and effects of ultrafine and fine particulate matter.

Slide courtesy of Mike Hannigan

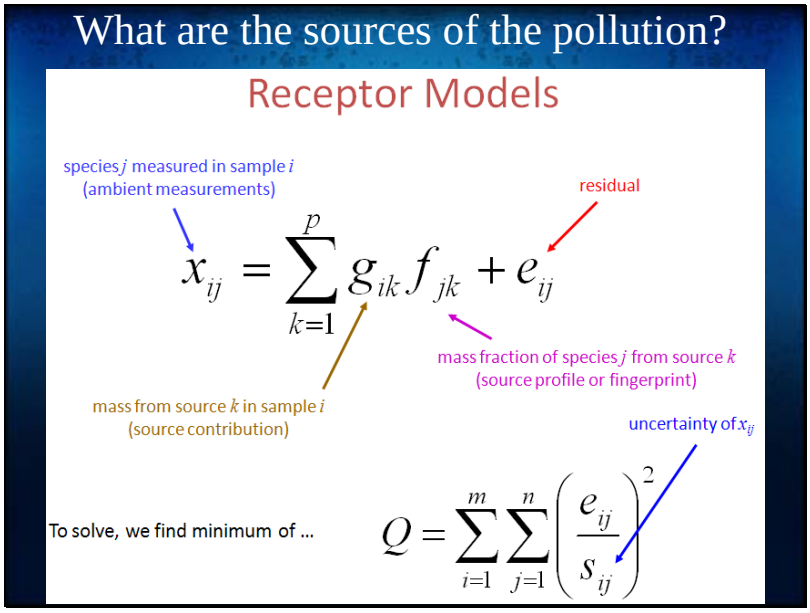
Slide 14



Ozone is associated with health effects, though increased mortality is not one of them. Conversely, fine particulate matter is associated with increased mortality. Sweat the small stuff!

Slide courtesy of Mike Hannigan

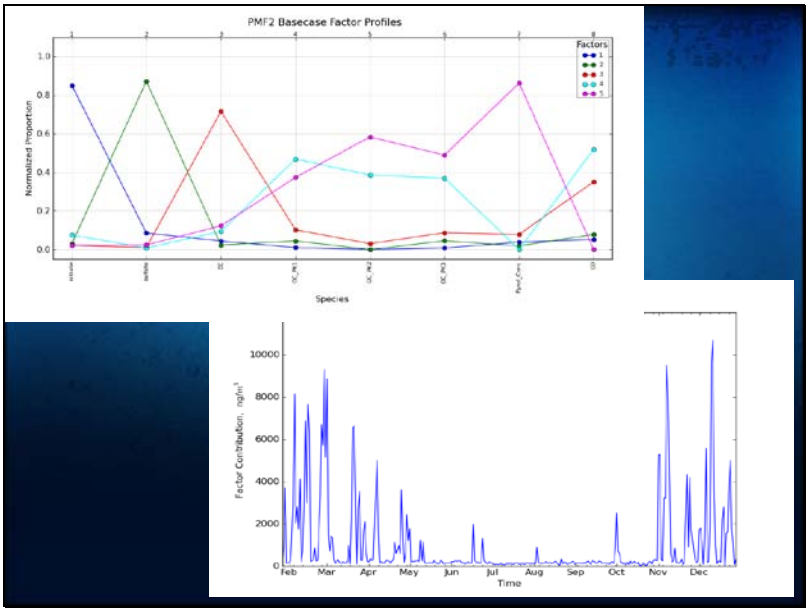
Slide 15



Receptor models used to apportion measured pollution to environmental sources (“factors” in the model). Elements of F and G are constrained to be non-negative. Non-negative matrix factorization was first developed by Pentii Paatero in the early 90s to handle this issue, later extended and now quite a useful tool in many fields. Model uncertainty: We have to choose p, the number of factors, we think can be extracted from the data. Hard to do a priori => explore different p values.

Note that you can weight specific measurements by specific uncertainties.

Slide 16



Dimensionality reduction.

Top graph: Rows of F, "factor loadings" or "profiles": what chemical species the factor accounts for.

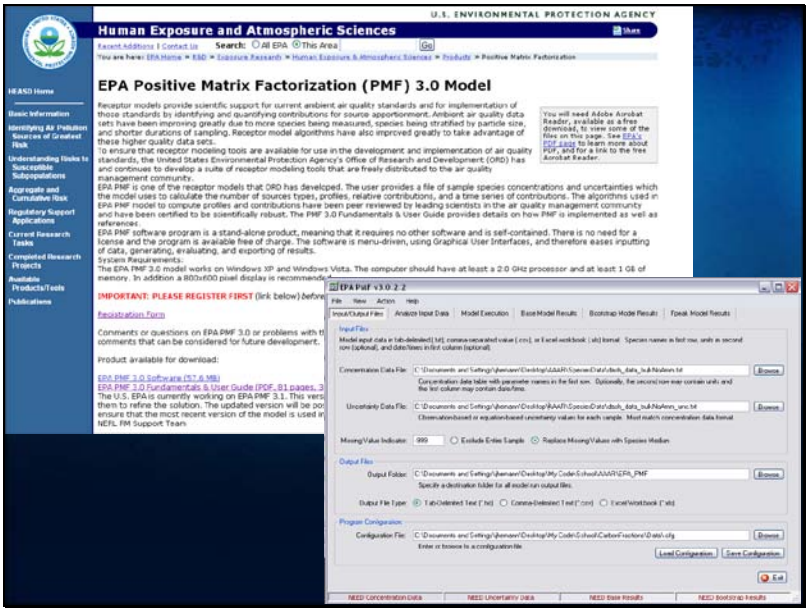
Bottom graph: Columns of G, "factor scores" or "contributions": what the factor is doing in time.

Thus, the measurement matrix X is factorized into constituent matrices characterizing

-How much a factor spits into the air
-What is in the stuff it spits out

Little sense of uncertainty can be gleaned from the model results though!

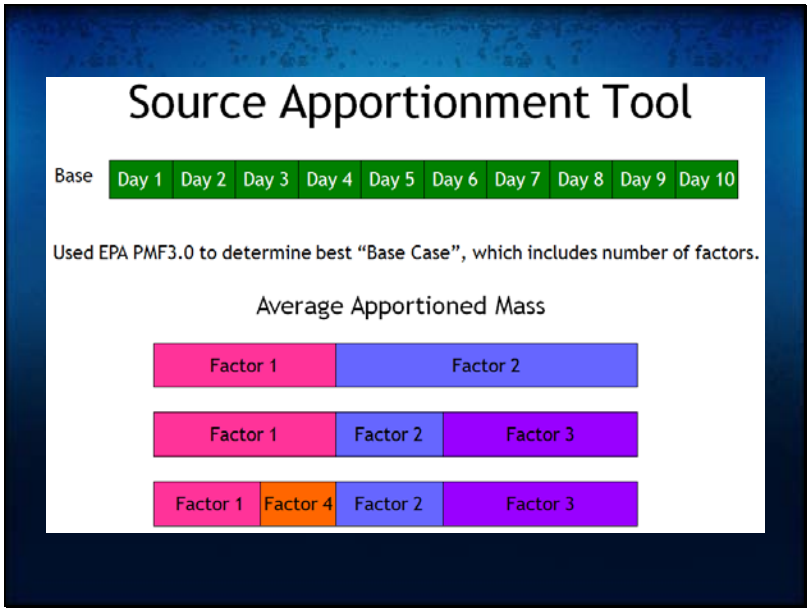
Slide 17



Tools exist to perform this pollution source apportionment. The US EPA provides a MATLAB-based tool called EPA PMF that is used by many practitioners modeling air pollution. Real-world impact: policy decisions are shaped by the results of our tools, but EPA PMF has some room for improvements…

http://www.epa.gov/heasd/products/pmf/pmf.html

Slide 18

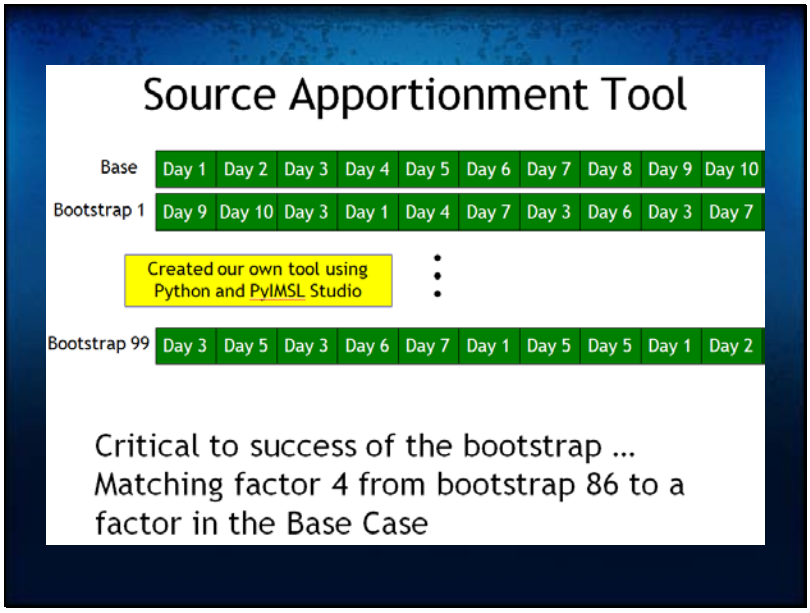


Experiment with different numbers of factors. How many factors can we reasonably extract from the data? How stable are the results if the data are perturbed?

In the middle row, assume Factor 1 looks like "automobile fuel combustion". If you add another factor to the solution, Factor 1 might split out to gas and diesel combustion. If you add too many factors though the mathematical factors may not have real-world analogues, and the degenerate case is just one factor per chemical species...

Slide courtesy of Mike Hannigan

Slide 19



Mathematical "factor 1" is row 1of F and column 1 of G, and say it looks like gasoline combustion. In a bootstrap solution this same mathematical factor might appear but just happen to be in row 2 of F, column 2 of G. We need to sort the rows/columns of the solutions such that "factor 1", i.e., row 1 and column 1 in every F,G matrix always corresponds to gasoline combustion.

Slide courtesy of Mike Hannigan

Slide 20

Analysis Steps...

1. Use Non-negative matrix factorization to factorize measurement matrix ***X*** into ***G*** (factor scores) and ***F*** (factor scores). This will be the "base case" model**.**
2. Block bootstrap resample measurement days (rows) in ***X*** to yield ***X****
3. Factorize ***X**** into ***G****,***F****
4. Use neural network or naive Bayes classifier to match factors in ***G**** and ***F**** with base case ***G*** and ***F*** (i.e. sort the columns of ***G****and the rows of ***F**** such that factor *i* always corresponds to the same column/row index in any given ***G***/***F*** matrix)
5. Repeat steps 2. through 4. 1,000 times
6. With pile of ***F*** and ***G*** matrices, compute descriptive statistics for each element, generate visualizations, etc

Step 1 is the computationally time consuming part.

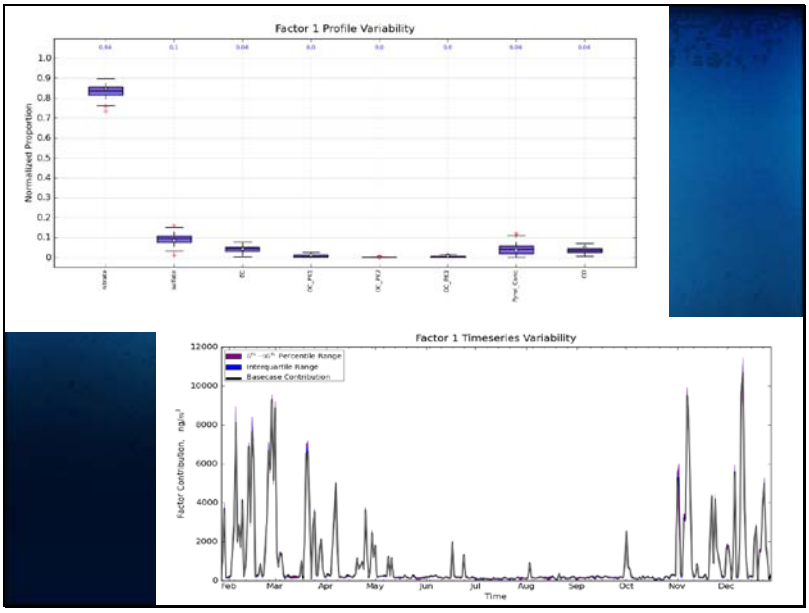
Step 2: Stationary Block Bootstrap. Mean found by looking at median p for AR(p) models fit to each pollutant species' time series.

Step 2: keep track of which days were resampled and where they showed up in the synthetic X!

Step 4 uses pattern classification algorithms in PyIMSL Studio

Embarrassingly parallel analysis

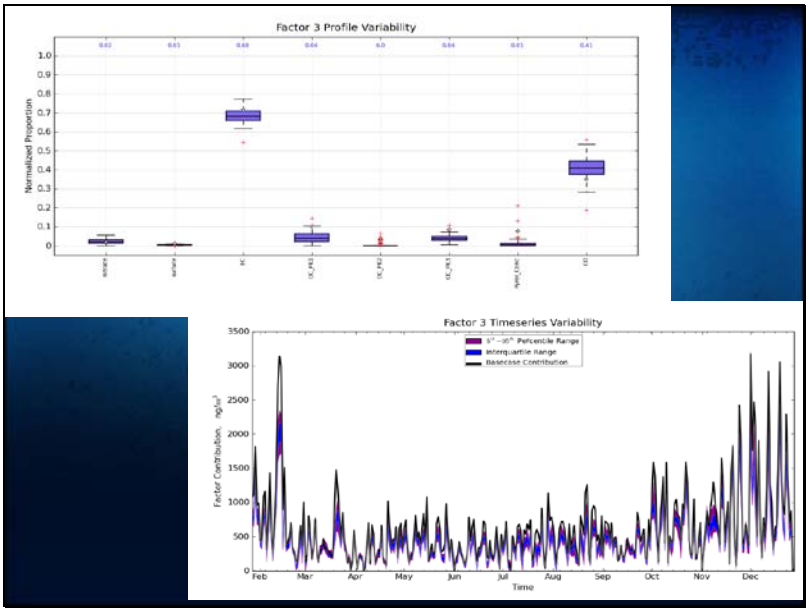
Slide 21



These results show bias and variability in the model results due to sampling error. This is what the huge simulation gets us!

This factor is what we might call the "nitrate factor"

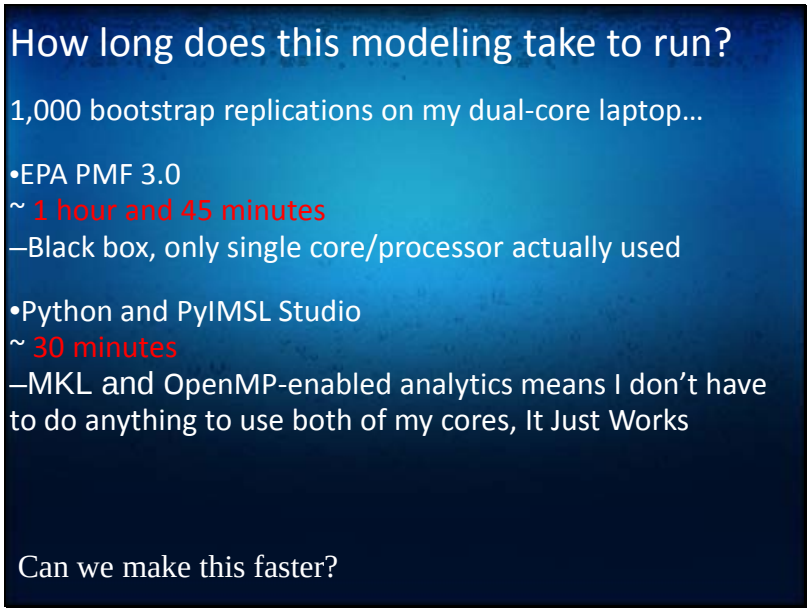
Slide 22



More variability and bias seen in Factor 3 than in Factor 1.

This factor is what we might call "diesel combustion", and we should question the base case model results for this factor due to the variability and bias we see under the bootstrap simulation.

Slide 23



Even though key algorithms might be already be multi-threaded, the entire analysis itself can be parallelized too.

Code can actually run slower using MKL and OpenMP due to thread overhead (keywords: OpenMP pragmas and nesting). Can't just blindly turn them on if you are concerned about speed, need to understand nature and scale of your problem. Automatic tuning of libraries on specific hardware, for specific problem types (e.g. ATLAS), is the wave of the future.

Slide 24

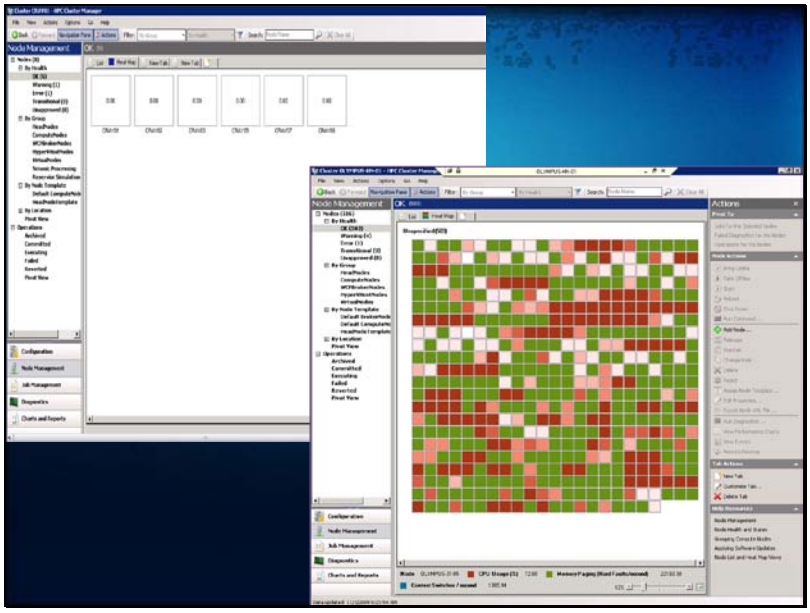
```
from IPython.kernel import client

#Set up each Python session on the clients...
mec = client.MultiEngineClient(profile='DASH')
mec.execute('import os')
mec.execute('import shutil')
mec.execute('import socket')
mec.execute('import parallelBlock')
mec.execute('reload(parallelBlock)')
mec.execute('from parallelBlock import parallelBlock')

#Task farm-out the 6 analysis steps...
tc = client.TaskClient(profile='DASH')
numReps = 1000
taskIDs = [ ]
for rep in xrange(1,numReps):
    t = client.MapTask(parallelBlock, args=[rep])
    taskIDs.append(tc.run(t))
tc.barrier(taskIDs)
results_list = [tc.get_task_result(tid) for tid in taskIDs]

for task, result in enumerate(results_list):
    #Unpack results from each iteration and do analysis/visualization
```

IPython to the rescue...

parallelBlock is just a Python function that performs the six analysis steps outlined a few slides back. Very little boilerplate needed to parallelize the simulation. Basically, we set up the environment on each engine in the cluster, map the six-steps analysis out to the engines, wait until the engines have performed the work, aggregate the results from each engine, and then process the results…

What I love about IPython (besides the interactive shell) is that while it supports multiple ways to do parallel computing, I can use it to quickly and easily parallelize “embarrassingly parallel” problems like I have in this context.

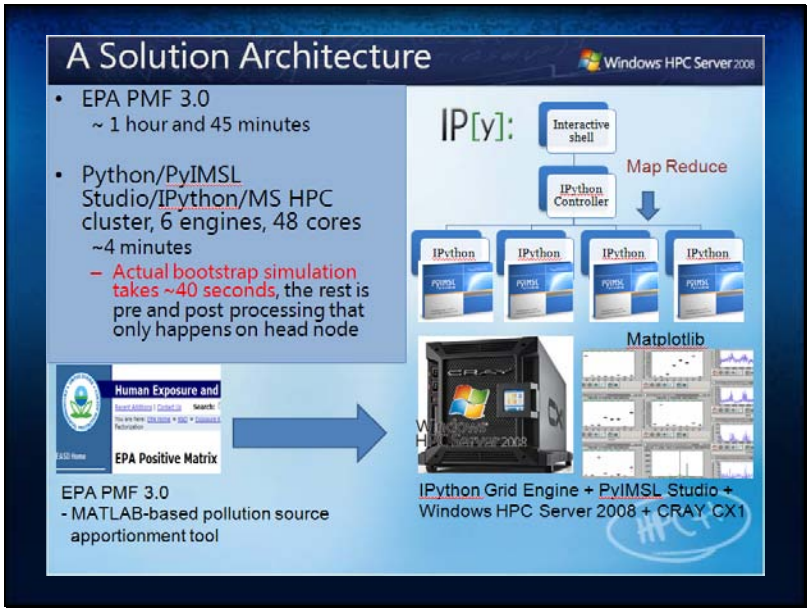
Slide 25



MS HPC Server makes administering cluster easy! Lots of 3rd party support, IPython parallel features dump xml configuration read in by HPC Server. I can remote desktop to each engine and browse the local file system, see a real-time heat map of processor load for each core, manage job schedules, etc, all in an intuitive interface. Great for someone like me who does not know much about the IT issues involved with managing a cluster.

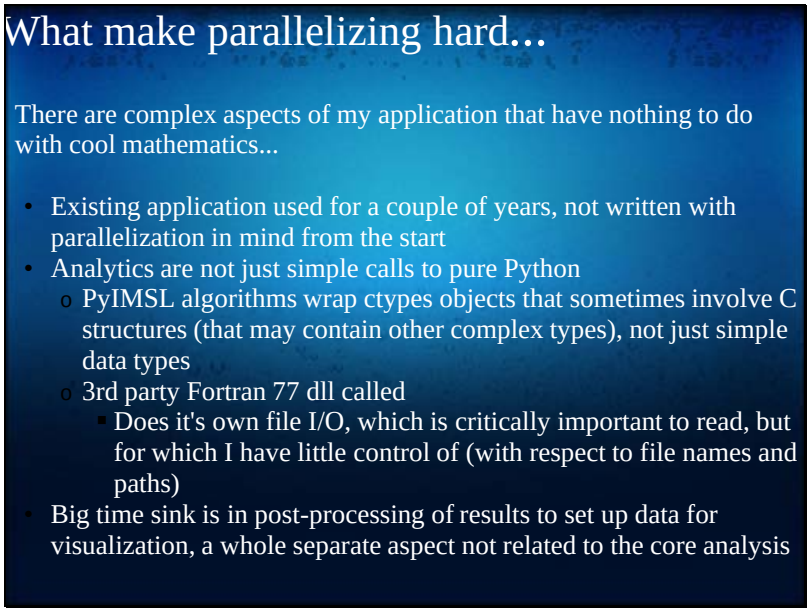
Lots of value in just taking advantage of the cores on one's desktop/laptop, HPC Server not needed. Code works the same, but different IPython profiles would be used to distribute the code on your dual-core laptop vs working with MS HPC Server to work over a cluster.

Slide 26



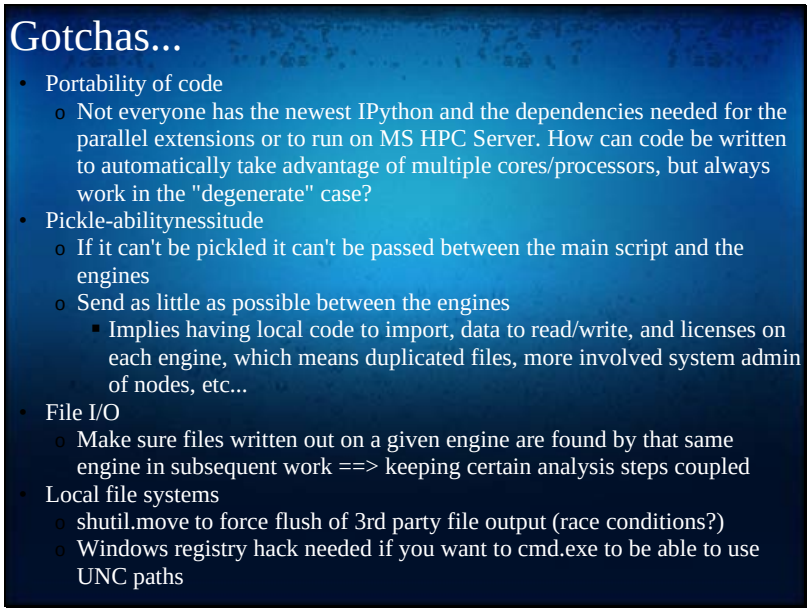
Slide courtesy of Wen-ming Ye, Technical Evangelist (HPC), Microsoft. Wen-ming got me access to the Cray CX1 cluster.

Slide 27



What happens when we move beyond simple examples of computing a matrix product?

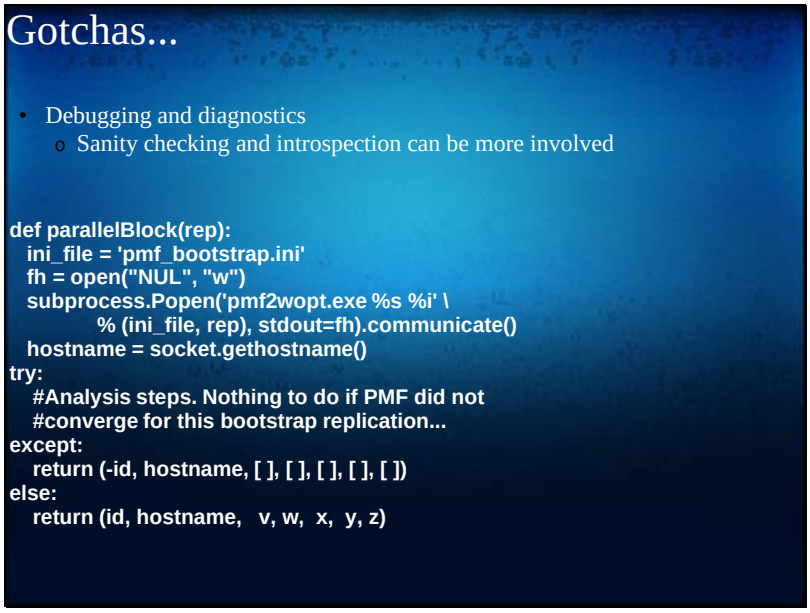
Slide 28



Bullet 1: Could be a simple try/except around importing IPython, but...

IPython 0.11 will help this issue, as will time: soon everyone will have multicore machines and be writing parallelized applications from the start.

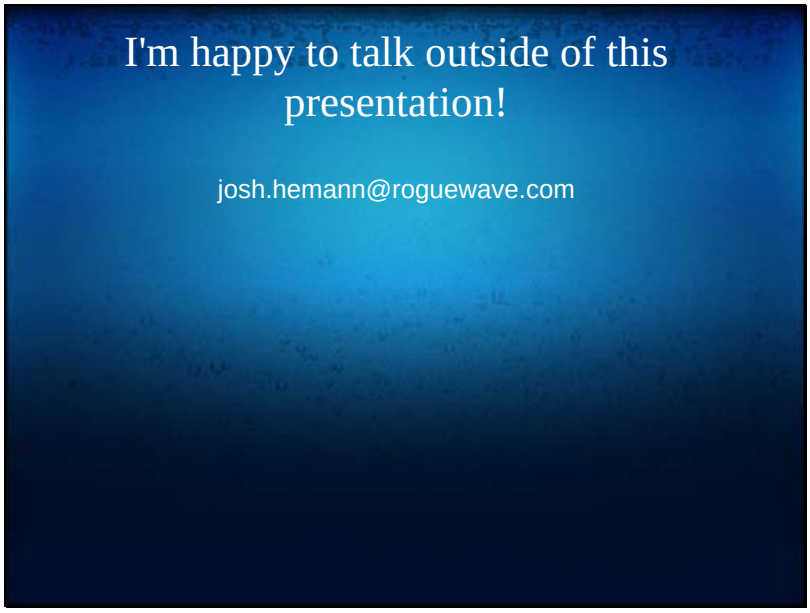
Slide 29



Whereas the non-parallel code used to just return a tuple of various results, now I have try/except blocks and also return information about the host and process ID, which is helpful for when I have to inspect a given engine's file system say, and track results across various processes.

I have to keep the PMF file I/O and subsequent analysis linked. engine ID is variable "id" in each engine's namespace

Slide 30


I'm happy to talk outside of this presentation!
josh.hemann@roguewave.com