Python in a Parallel Environment

Dave Grote – LLNL & LBNL

NUG2013 User Day

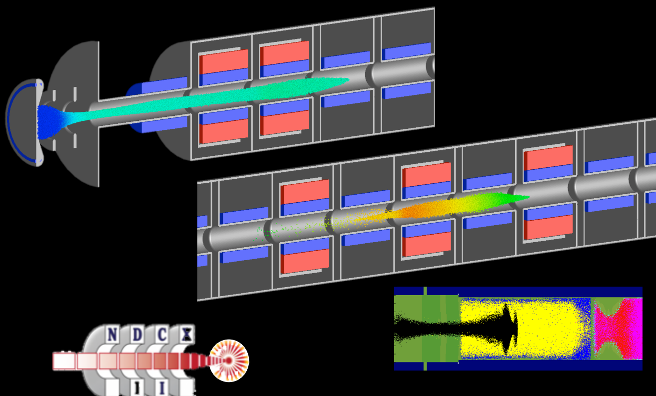
Wednesday, February 15, 2013

Outline

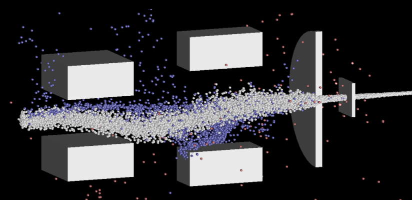
- Why we use Python
- How we use Python
- Parallel Python with pyMPI
- Our graphics model with Pygist
- Parallel Python drawbacks and resolutions
  - Start up time
  - Static building
- Conclusions

Warp is a framework for particle accelerator modeling

HIF/HEDP accelerators

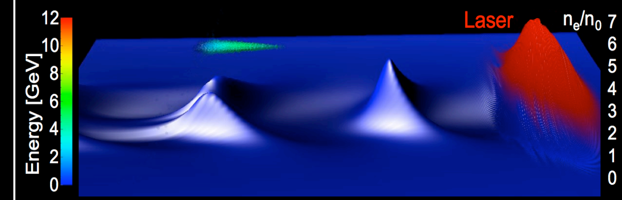


Multi-charge state beams



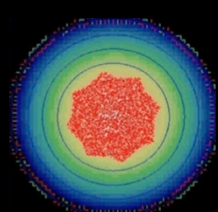
LEBT – Project X

Laser plasma acceleration

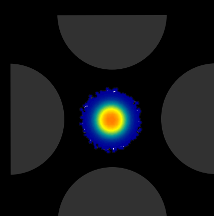


BELLA

Particle traps



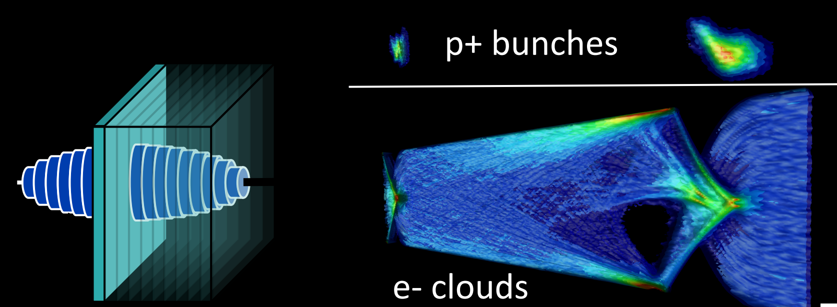
Alpha anti-H trap



Courtesy H. Sugimoto

Paul trap

Electron cloud studies



Why Python?

- Python is a high level, interpreted and interactive language
- This allows for flexible and rapid application development
  - It is easy to develop, test, and apply scripts, with quick turn around
  - The language is full featured, allowing high level programming when needed (object orientation for example)

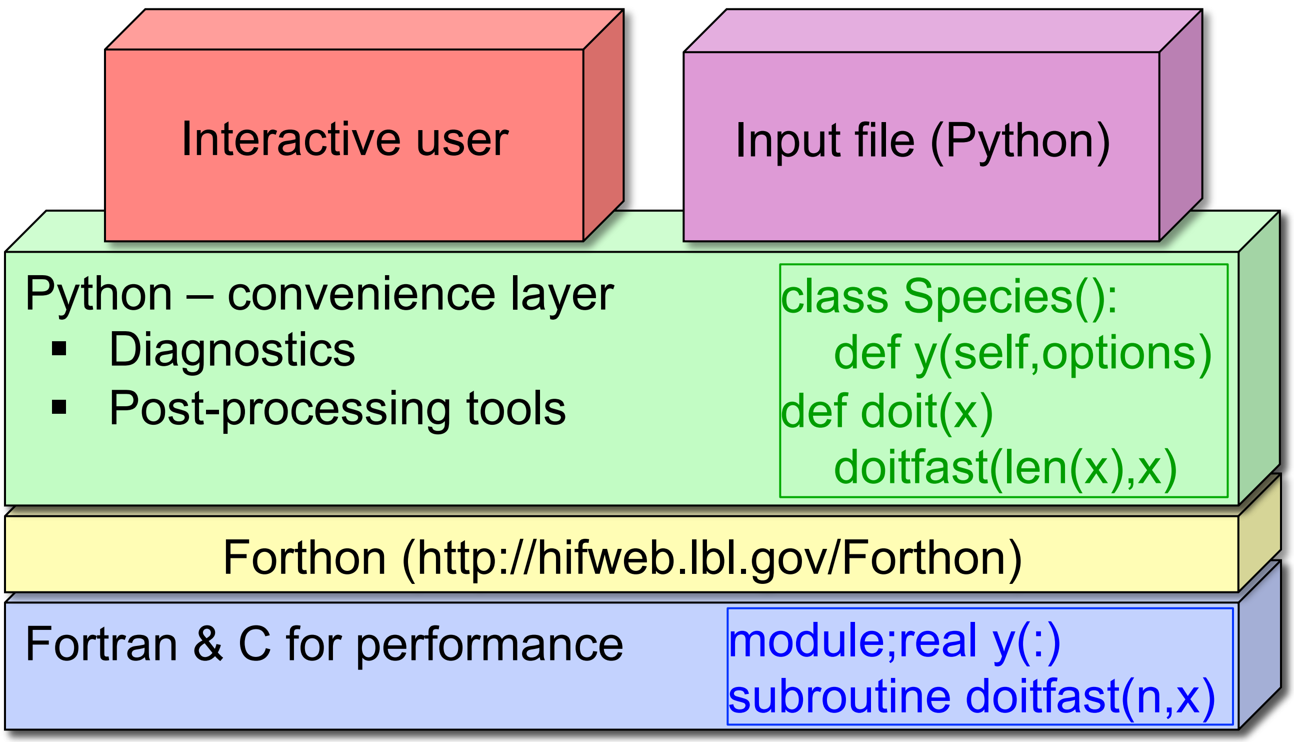
- Allows "steering" of simulations, via scripting and/or interactivity
- Interactivity allows on-the-fly diagnostics and post processing

- It is reasonably fast
- …and when not fast enough, can easily connect to compiled code

- Large available library of standard and third party packages
  - The most important (for us) is *numpy* for fast array operations

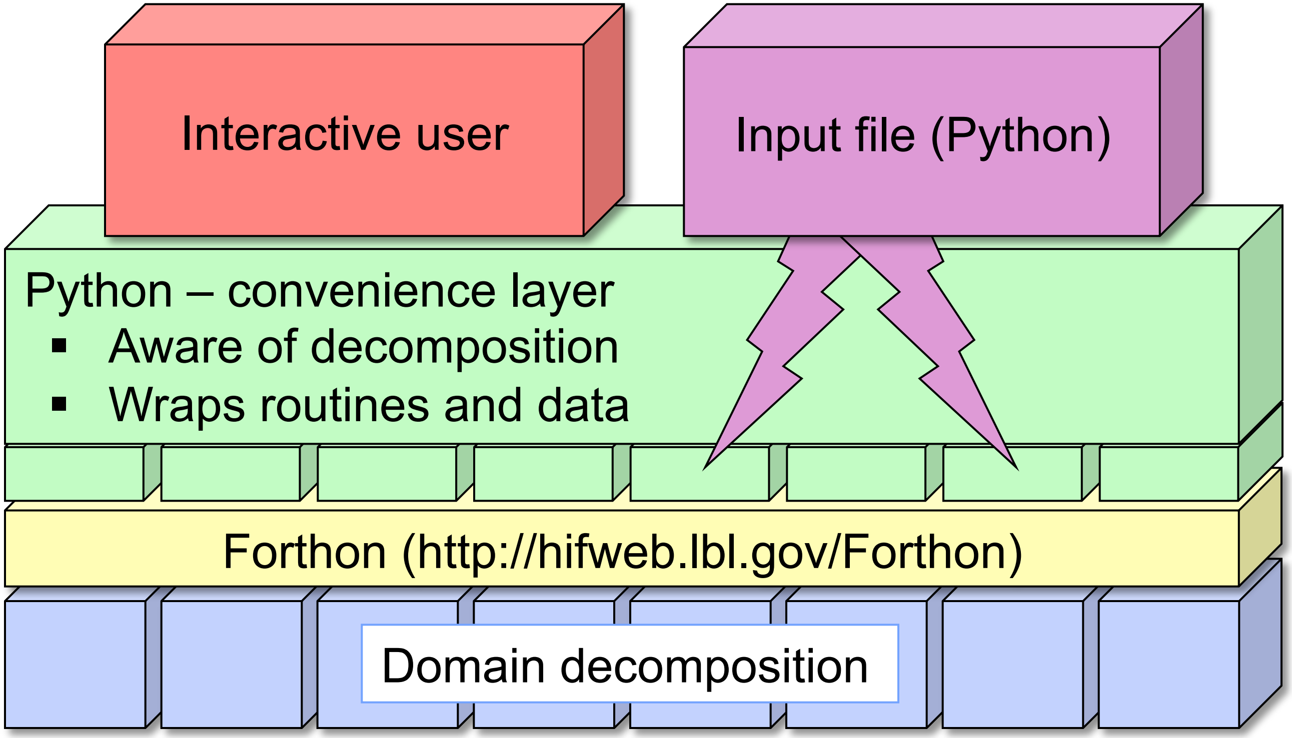
Python provides the user interface to Warp

- Warp is a framework for particle accelerator modeling



Parallel Python

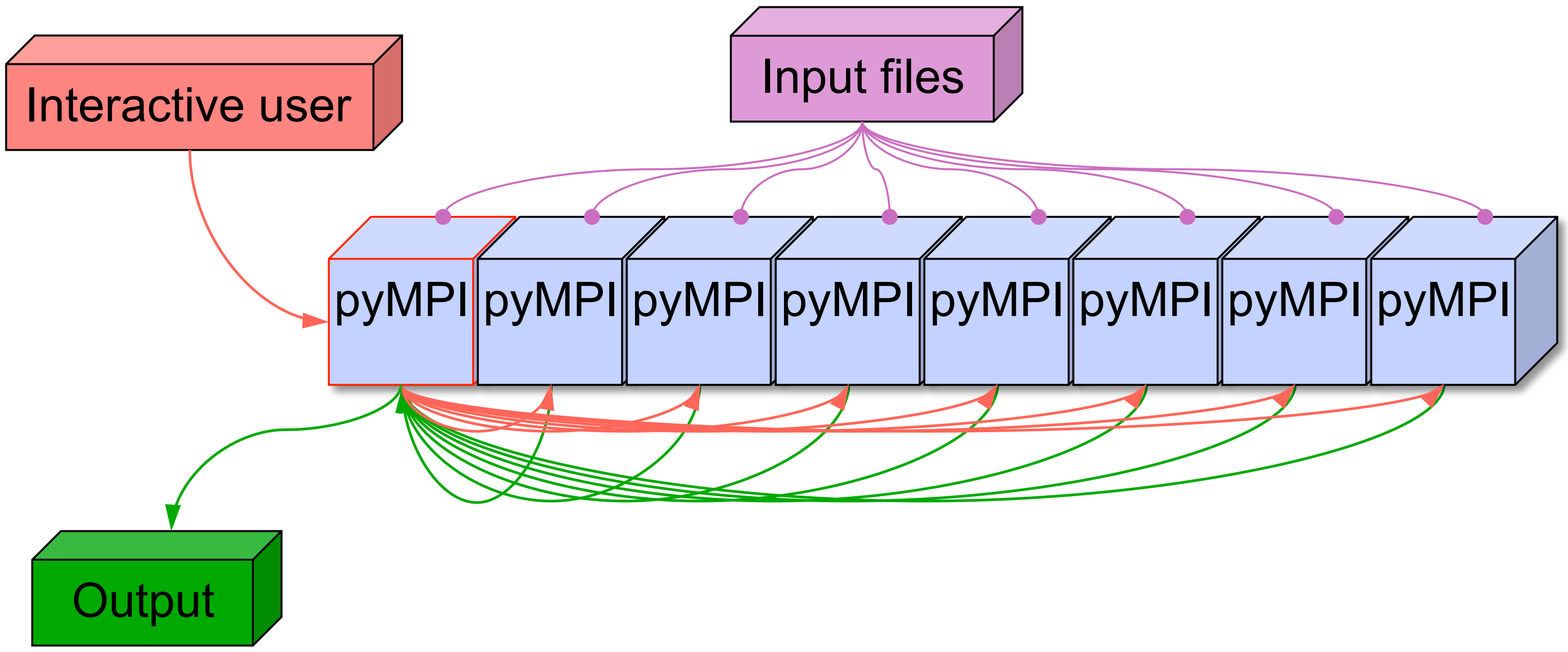
- The goal is to have the parallelism invisible to the user



- However, complete invisibility is not quite possible (nor desirable)
  – Sometimes low level access is needed for flexibility and performance
  – Impossible to eliminate gotchas when user can access everything

Parallel Python with pyMPI

- Originally developed by Pat Miller (at LLNL)
- Serves two purposes:
  - Provides Python level interface to MPI routines
  - Allow interactivity in a parallel environment

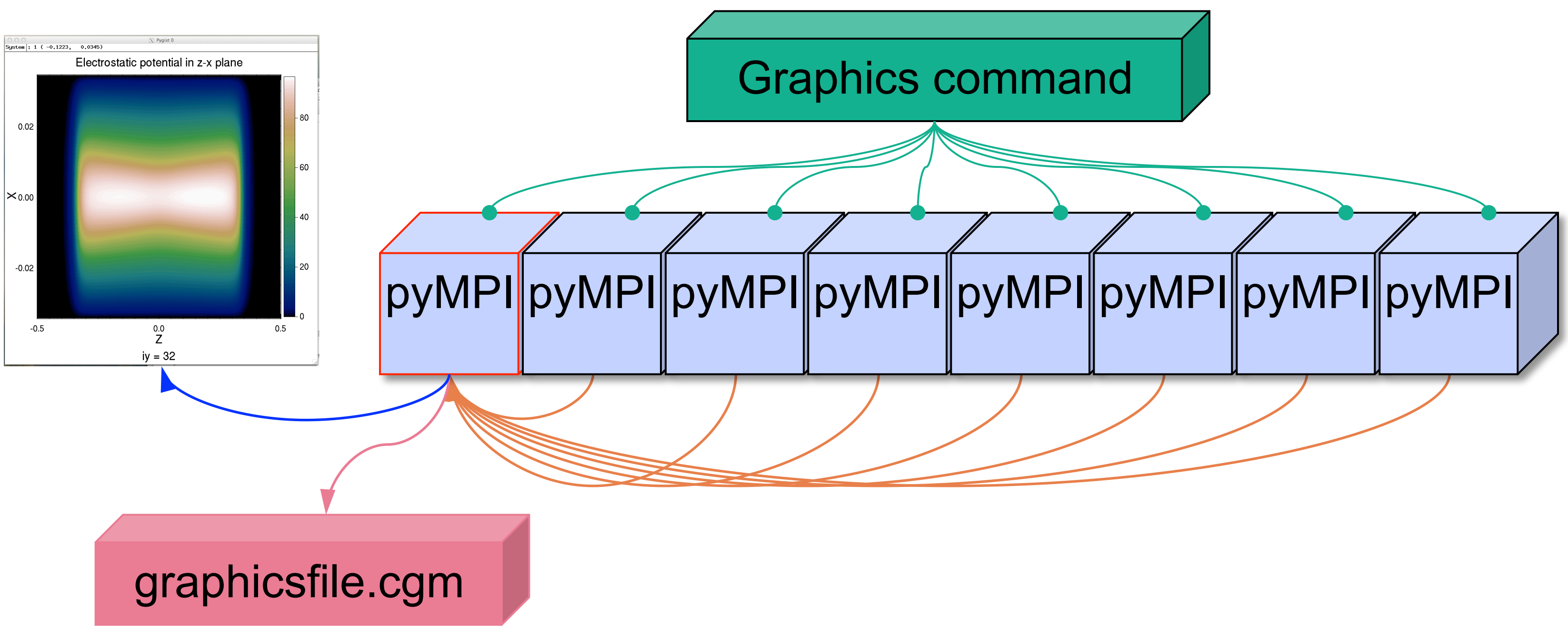


Parallel build only somewhat more complicated

- Like the Python executable, pyMPI only needs to be built once
- For Warp, build includes extra parallel code – handled by the Makefile
- Python distutils used the same way as in serial
  - Builds the shared object file
  - Only complication is adding MPI libraries

Graphics on the fly – pygist

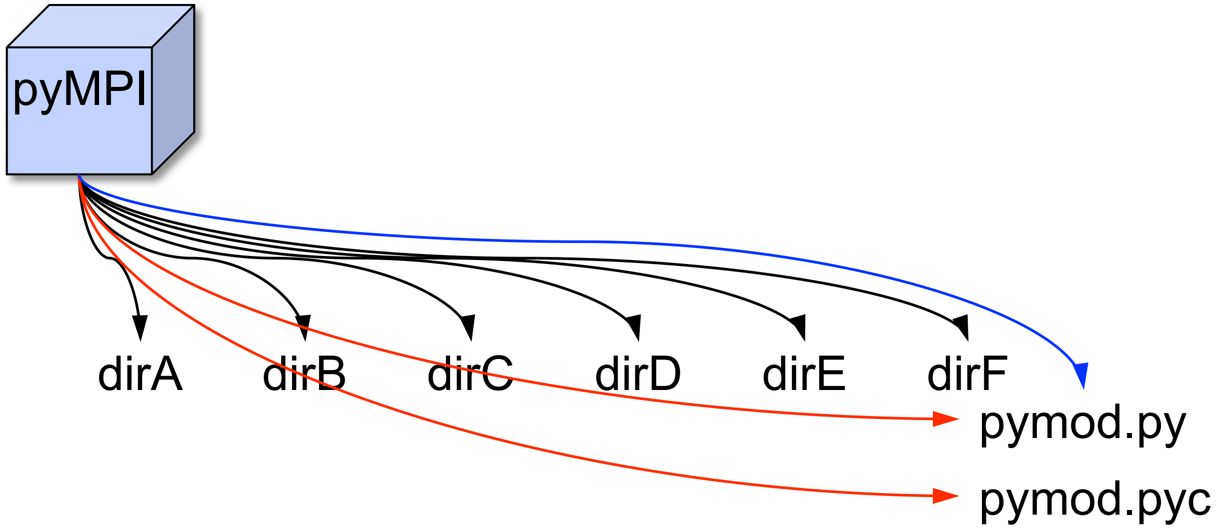
- Our primary way of working is to produce graphics in line, while the simulation is running
  - Can easily track the progress of the simulation
  - Efficient since data is immediately available
  - Reduces amount of data saved



Major drawback – import doesn't scale

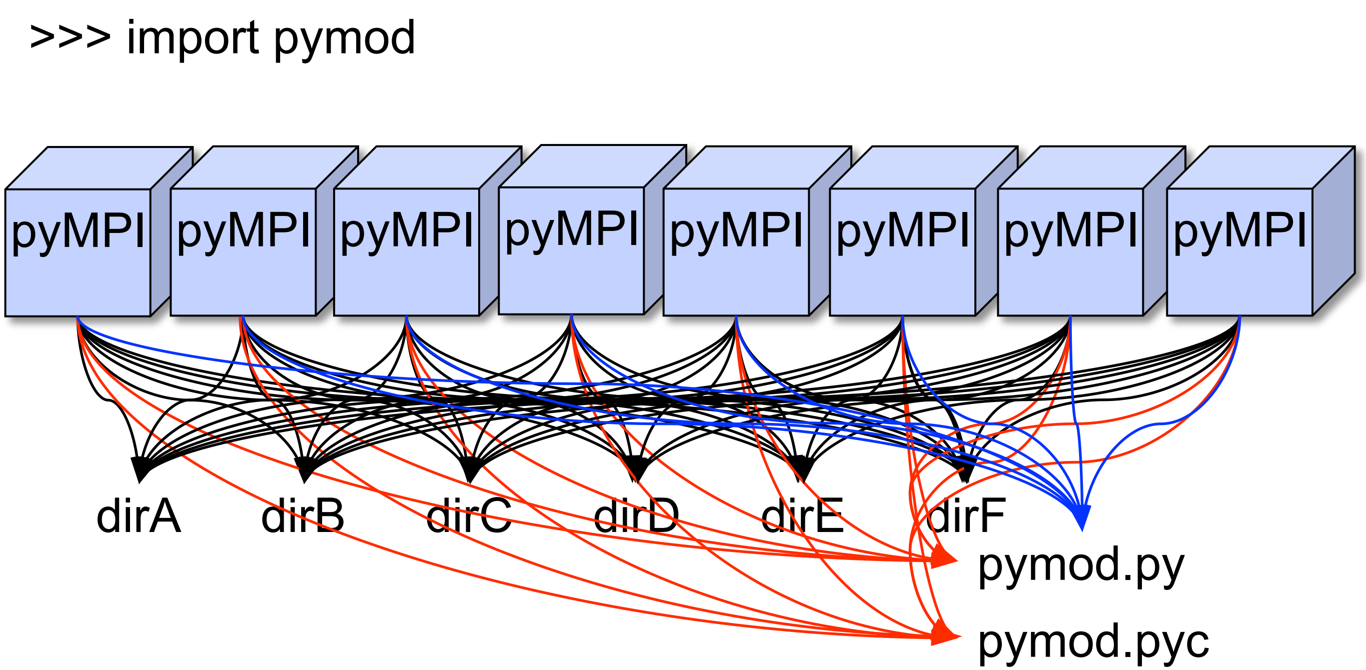
- Every processor does it's own import

>>> import pymod



Major drawback – import doesn’t scale

- Every processor does it’s own import

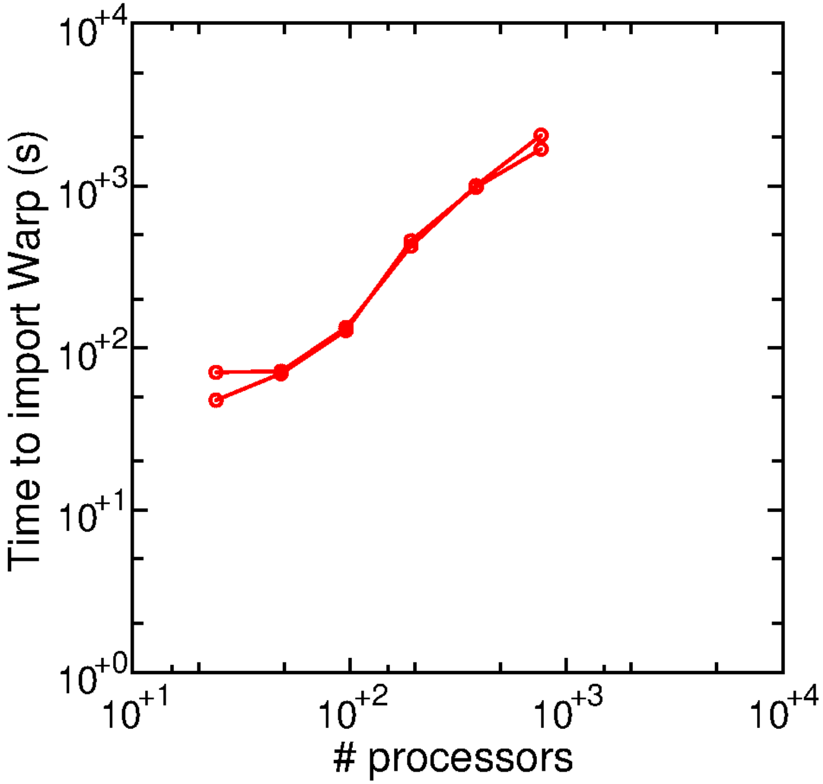


- “import warp” leads to importing over 225 modules

Major drawback – import doesn't scale

- On one Hopper node, the start up time is about a minute and is ignorable
- On a large run though, the start up time can eat hours

This is the time it takes for "import warp", the first statement in any Warp input file.



Avoiding the bottleneck

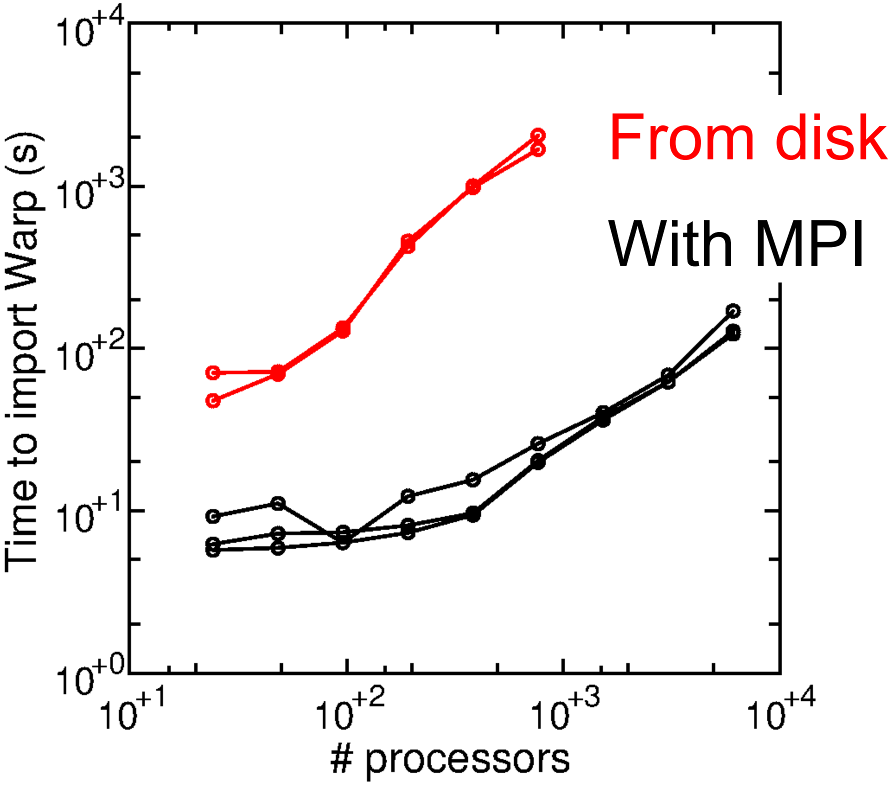
- Much of the time is spent checking the status of files
- Some of the time is spent reading in the files

- There are various possible remedies
  - Moving Python to the scratch disk improved the times, but not enough
  - Precopy files to local disks to spread the load
    - DLCache/FMCache packages do this
    - Requires trial run and pre- and post- runs to setup cache
  - Use MPI

Fastest solution is to use MPI

- Communicating data through MPI is much faster than via disk
- Solution is to have one processor do the work and broadcast to the rest
  - First processor finds the correct file and reads it in
  - The file info and contents are broadcast via MPI

- Caveats:
  - Python needs to be hacked
  - What about shared objects?



Hacking Python

- Some of the speed up comes modifying the file search
  - In a number of places in import.c calls to stat were wrapped
  - The results are broadcast
- I also wrapped the reading in of imported scripts
  - In various places I wrapped fopen, fclose, getc etc
  - This gave most of the rest of the speed up
- Since I was already hacking the code, I went overboard
  - I wrapped the reading in the input file on the command line
  - I wrapped the reading in of files from execfile
  - These gave small additional speed up, but are not necessary
- One possible limitation:
  - The code must be SPMD when doing imports

Static version of Python

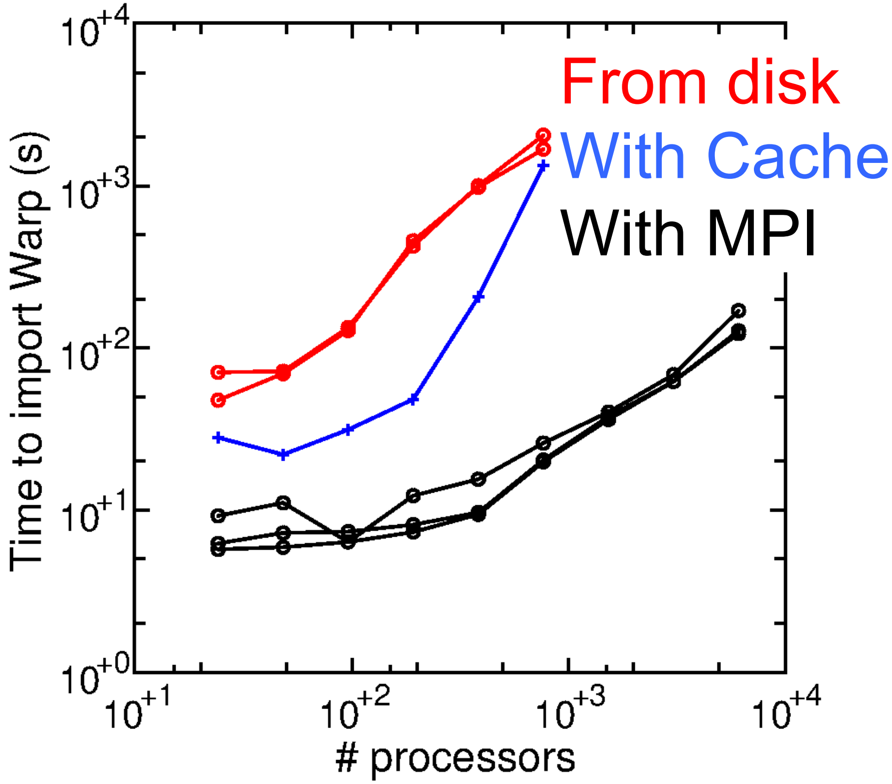
- There are two issues with dynamic loading:
  - Startup time
  - Not always supported in HPC environments
- Unfortunately, static loading is not supported in Python for extensions
  - More hacking is needed
- Method based on what was developed for GPAW
  - distutils modified to build static libraries and put them in the right place
  - The dynamic loader is replaced with code that returns builtins
  - Modules/Setup needs to be modified by hand to add module info
- Once this is done, it is essentially indistinguishable to the user

Can the bottleneck be fixed at the Python level?

- To a degree, yes
- Major bottle neck is finding and verifying the correct file
- Asher Langton of LLNL wrote alternatives
  - Python allows modification of the import mechanism
  - The locations of modules are cached (and possibly broadcast)
  - Finding the module is done using the cache
  - Each processor reads in the appropriate file
- This can make an improvement in the import time
  - He reported a reduction from 5.5 hours to 6 minutes on 32k processor Blue Gene

However, I don't see the same speed up

- Running on Hopper
- It is not clear why



Conclusions

- Python has proven to be an extremely valuable tool for computational science
- In a parallel environment, it can become even more valuable for easy of use, simplicity, and convenience
- However, there are drawback that must be dealt with
- The problems are solvable, but sometimes require getting your hands dirty
- …but the payoff is well worth it

- This version of Python is available on Hopper, if there is interest (but with minimal support)

Extras

Python provides the user interface to Warp

- Warp is a framework for particle accelerator modeling
- Lower level is Fortran and C
  - Compiled for performance
  - This is wrapped by Forthon (http://hifweb.lbl.gov/Forthon)
  - Subroutines are callable and data is accessible from Python
- Middle level is Python
  - Higher level wrappers around the compiled data
  - Wrappers around the compiled routines for simpler, convenient interface
  - Extensive diagnostics and post processing tools
- Top level, the user interface, is Python
  - Input file is Python
  - Interactivity

Parallel Python

- The goal is to have the parallelism invisible to the user
  - Identical input files (Python scripts) for serial and parallel
  - Interactivity the same as in serial
- Much can be done at the middle level to hide the parallelism
  - Provides wrappers around domain decomposed data
  - Provides high level routines that automatically handle the parallelism
- However, invisibility is not quite possible
  - Sometimes low level access is needed for flexibility and performance
  - Impossible to eliminate gotchas and lockups when user can access everything

Parallel Python with pyMPI

- Originally developed by Pat Miller (at LLNL)
- Serves two purposes:
  - Provides Python level interface to MPI routines
  - Allow interactivity in a parallel environment
- pyMPI is a separate executable that incorporates Python
- pyMPI runs on every processor and all execute the same Python code
- First processor handles interactivity
  - All input read in by first processor and sent to all other processors
  - Output can be controlled – either only from first processor, all processors or a mix
- Extensive (though not complete) wrapping of MPI
  - Point-to-point and global operations
  - Any "pickle-able" object can be sent

Graphics on the fly – pygist

- Our primary way of working is to produce graphics in line, while the simulation is running
  - Can easily track the progress of the simulation
  - Efficient since data is immediately available
  - Reduces amount of data saved
- Matches Python's scripting and interactivity
  - The input file can make arbitrary plots
  - The graphics window is interactive (zooming and panning)
- Matches the parallelism
  - Computations done in parallel (down selections, slicing, reductions etc)
  - Only a small amount of data is written out by the master