1 Python in HPC

1.1 Supercomputing 2012

Presenters:
Andy R. Terrel, PhD
Texas Advanced Computing Center
University of Texas at Austin

Travis Oliphant, PhD
Continuum Analytics

Aron Ahmadia, PhD
Supercomputing Laboratory
King Abdullah University of Science and Technoglogy

1.2 Updated Tutorial

These presentation materials are being continuously updated as we refine and improve our demonstrats. To get the latest version of this tutorial you can:

1) Download a zip or tar ball from the github SC2012 tag:

   ```
   wget –no-check-certificate https://github.com/aterrel/HPCPythonSC2012/zipball/SC2012
   ```

2) Checkout from git

   ```
   git clone https://github.com/aterrel/HPCPythonSC2012.git
   ```

3) View the html version on nbviewer.

4) As a last resort, head to https://github.com/aterrel/HPCPythonSC2012 for updated instructions (see the README at the bottom of the page).

1.3 Interacting with the Tutorial Slides

This tutorial is an interactive worksheet designed to encourage you to try out the lessons during the demonstration. If you are looking at the pdf version, we encourage you to download the updated version (see previous slide) and try the interactive version.

To run the interactive version, you need a good Python environment including:

- IPython version >= 13.0
- Numpy version >= 1.5
- Scipy
- Matplotlib

Move to the directory containing the tarball and execute:

```
$ ipython notebook --pylab=inline
```

We heartily endorse the Free Enthought Python Distribution.

1.4 Acknowledgements

- Much of this tutorial adapts slide material from William Gropp, University of Illinois
- mpi4py is a Cythonized wrapper around MPI originally developed by Lisandro Dalcin, CONICET

1.5 The Free Lunch is Over

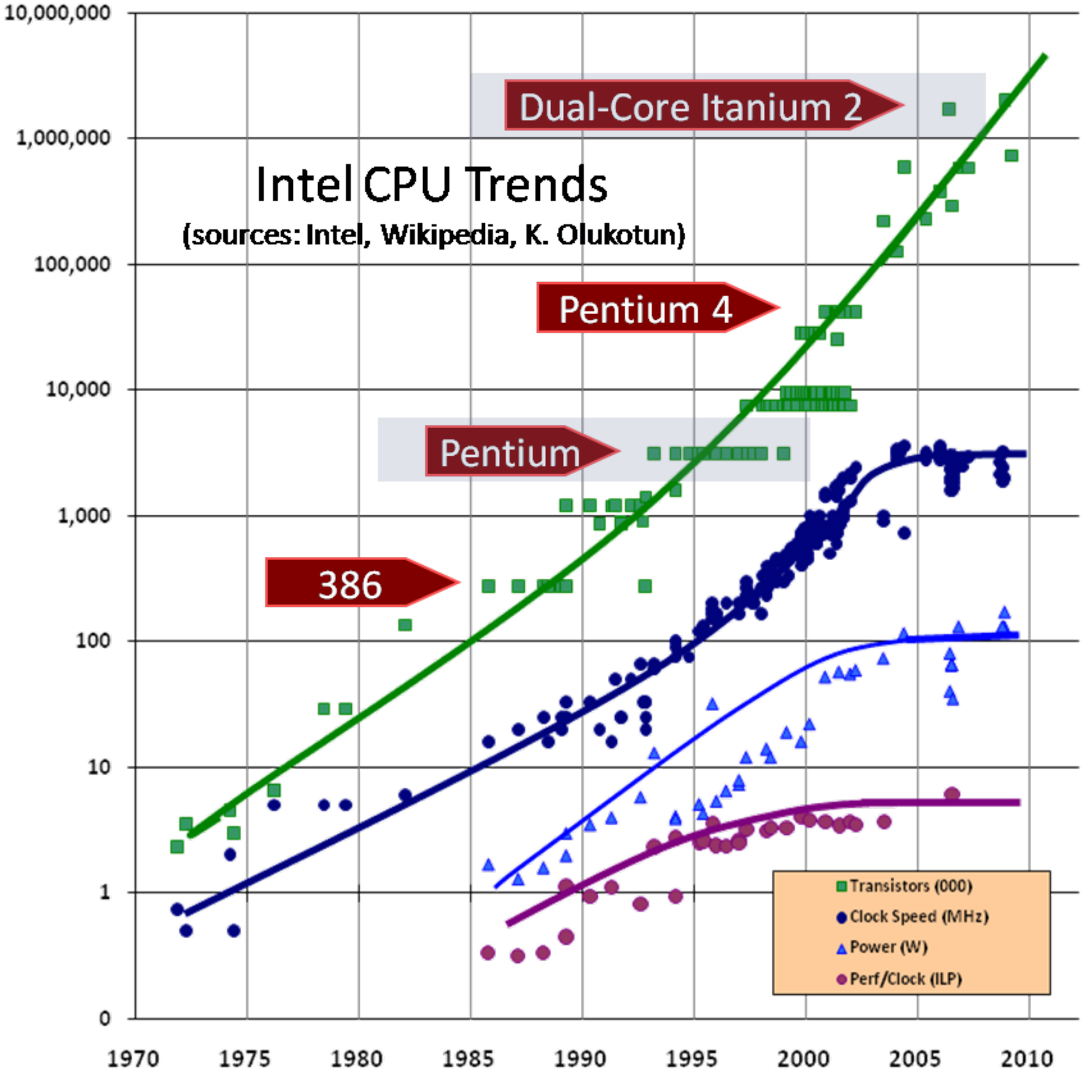
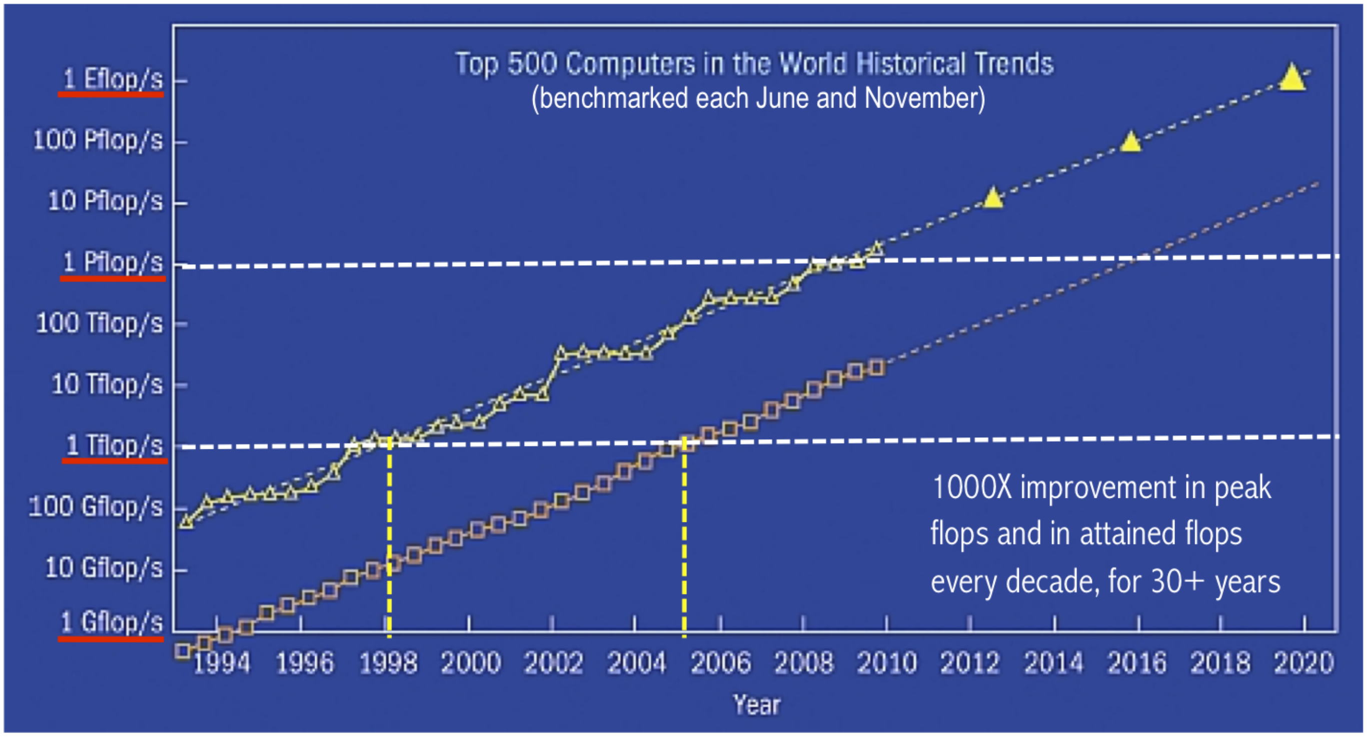


Figure 1: freelunch

2 The Multiple Forms of Parallelism

- **instruction** - multiple program instructions are simultaneously dispatched in a pipeline or to multiple execution units (superscalar)
- **data** - the same program instructions are carried out simultaneously on multiple data items (SIMD)
- **task** - different program instructions on different data (MIMD)
- **collective** - single program, multiple data, not necessarily synchronized at individual operation level (SPMD)

2.1 The Rise of Manycore



c/o *SciDAC Review* 16, February 2010

Figure 2: Rise of Manycore

3 Parallel Architectures

- All modern supercomputing architectures are composed of distributed memory compute nodes connected over toroidal networks
- close mapping to physically n-dimensional problems
- efficient algorithms exist for local point-to-point and collective communications across toroids
- We expect to see higher dimensional interconnects and power-efficient networks that consume less power when not sending traffic

4 Parallel Programming Paradigms

- a parallel programming paradigm is a specific approach to exploiting parallelism in hardware
- many programming paradigms are very tightly coupled to the hardware beneath!
- CUDA assumes large register files, Same Instruction Multiple Thread parallelism, and a mostly flat, structured memory model, matching the underlying GPU hardware
- OpenMP exposes loop level parallelism with a fork/join model, assumes the presence of shared memory and atomics
- OpenCl tries to generalize CUDA, but still assumes a 'coprocessor' approach, where kernels are shipped from a master core to worker cores

4.1 The Message Passing Model

- a process is (traditionally) a program counter for instructions and an address space for data
- processes may have multiple threads (program counters and associated stacks) sharing a single address space
- message passing is for communication among processes, which have separate address spaces
- interprocess communication consists of
- synchronization
- movement of data from one process's address space to another's

4.2 Why MPI?

- **communicators** encapsulate communication spaces for library safety
- **datatypes** reduce copying costs and permit heterogeneity
- multiple **communication modes** allow more control of memory buffer management
- extensive **collective operations** for scalable global communication
- **process topologies** permit efficient process placement, user views of process layout
- **profiling interface** encourages portable tools

It Scales!

4.3 Python + MPI Does It Scale?

- This and next slide courtesy the **PyClaw** project: http://numerics.kaust.edu.sa/pyclaw/
- See also **GPAW**: https://wiki.fysik.dtu.dk/gpaw/

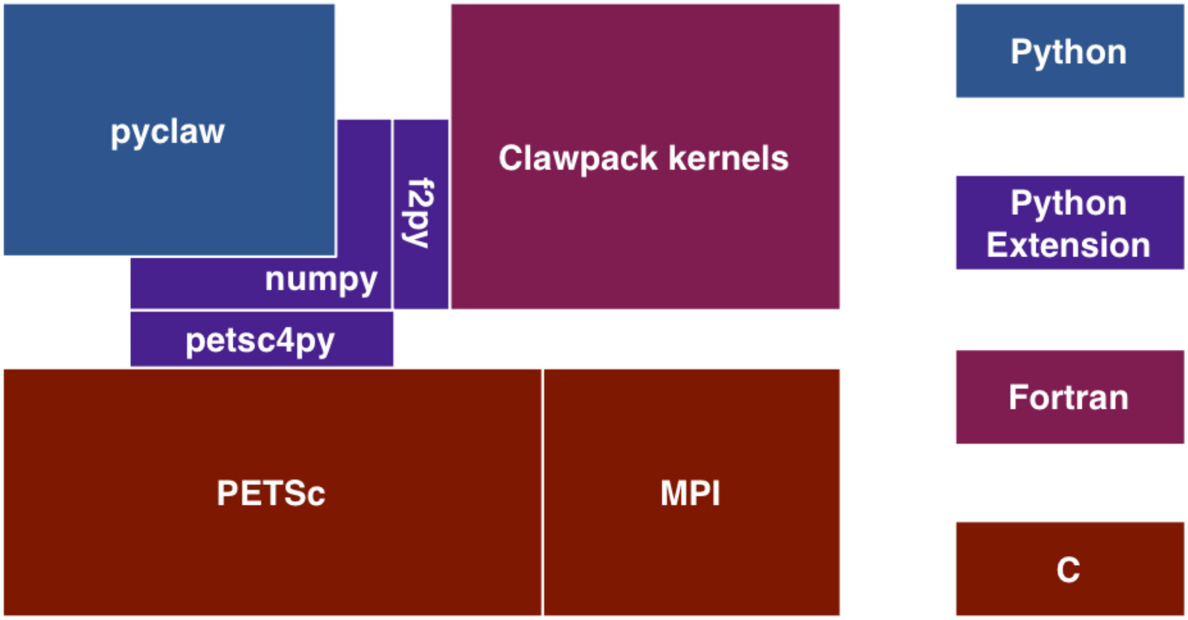
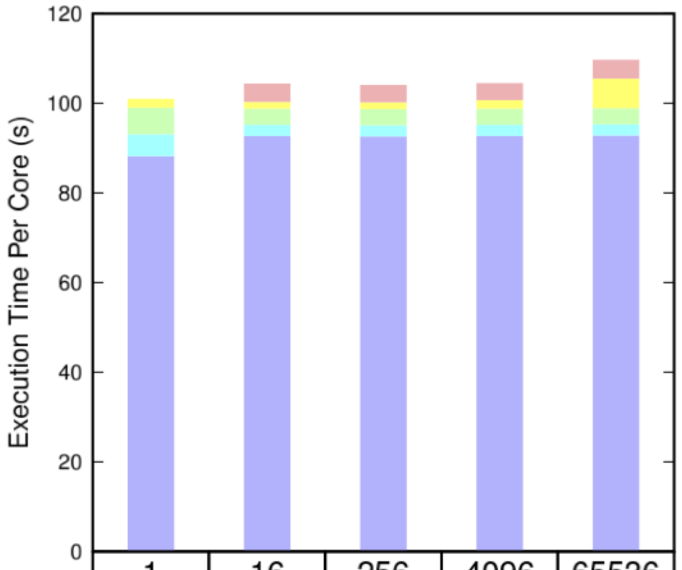


Figure 3: PyClaw Architecture

4.4 The Scalability of Python + MPI



	1	16	256	4096	65536
CFL Reduce	0.001	4.1	3.9	3.8	4.2
Parallel Initialization	2	1.5	1.5	1.9	6.6
Global to Local	5.9	3.6	3.6	3.6	3.6
Local to Global	4.9	2.5	2.5	2.5	2.5
Serial Computations	88.2	92.7	92.6	92.7	92.8
Total Efficiency	1.0	0.97	0.97	0.97	0.92

Number of Cores

Figure 4: Scalability

4.5 MPI - Quick Review

- processes can be collected into **groups**
- each message is sent in a **context**, and must be received in the same context
- a **communicator** encapsulates a context for a specific group
- a given program may have many communicators with any level of overlap
- two initial communicators
- `MPI_COMM_WORLD` (all processes)
- `MPI_COMM_SELF` (current process)

4.6 Communicators

- processes can be collected into **groups**
- each message is sent in a **context**, and must be received in the same context
- a **communicator** encapsulates a context for a specific group
- a given program may have many communicators with any level of overlap
- two initial communicators
- `MPI_COMM_WORLD` (all processes)
- `MPI_COMM_SELF` (current process)

4.7 Datatypes

- the data in a message to send or receive is described by address, count and datatype
- a datatype is recursively defined as:
- predefined, corresponding to a data type from the language (e.g., `MPI_INT`, `MPI_DOUBLE`)
- a contiguous, strided block, or indexed array of blocks of MPI datatypes
- an arbitrary structure of datatypes
- there are MPI functions to construct custom datatypes

4.8 Tags

- messages are sent with an accompanying user-defined integer tag to assist the receiving process in identifying the message
- messages can be screened at the receiving end by specifying the expected tag, or not screened by using `MPI_ANY_TAG`

4.9 MPI Basic (Blocking) Send

```
int MPI_Send(void* buf, int count, MPI_Datatype type,
int dest, int tag, MPI_Comm comm)
```

Python (mpi4py)

`Comm.Send(self, buf, int dest=0, int tag=0) Comm.send(self, obj=None, int dest=0, int tag=0)`

4.10 MPI Basic (Blocking) Recv

```
int MPI_Recv(void* buf, int count, MPI_Datatype type,
int source, int tag, MPI_Comm comm, MPI_Status status)
```

Python (mpi4py)

```
comm.Recv(self, buf, int source=0, int tag=0, Status status=None)
comm.recv(self, obj=None, int source=0, int tag=0, Status status=None)
```

4.11 Synchronization

```
int MPI_Barrier(MPI_Comm comm)
```

Python (mpi4py)

```
comm.Barrier(self) comm.barrier(self)
```

4.12 Timing and Profiling

the elapsed (wall-clock) time between two points in an MPI program can be computed using MPI_Wtime:

```
t1 = MPI.Wtime() t2 = MPI.Wtime() print("time elapsed is:  %e\n" % (t2-t1))
```

Send/Receive Example (lowercase convenience methods)

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
   data = {’a’: 7, ’b’: 3.14}
   comm.send(data, dest=1, tag=11)
elif rank == 1:
   data = comm.recv(source=0, tag=11)
```

Send/Receive Example (MPI API on numpy)

```python
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# pass explicit MPI datatypes
if rank == 0:
   data = numpy.arange(1000, dtype=’i’)
   comm.Send([data, MPI.INT], dest=1, tag=77)
elif rank == 1:
   data = numpy.empty(1000, dtype=’i’)
   comm.Recv([data, MPI.INT], source=0, tag=77)

# or take advantage of automatic MPI datatype discovery
if rank == 0:
   data = numpy.arange(100, dtype=numpy.float64)
   comm.Send(data, dest=1, tag=13)
elif rank == 1:
   data = numpy.empty(100, dtype=numpy.float64)
   comm.Recv(data, source=0, tag=13)
```

4.13 Broadcast/Reduce

```
int MPI_Bcast(void *buf, int count, MPI_Datatype type,
int root, MPI_Comm comm)
```

Python (mpi4py)

`comm.Bcast(self, buf, int root=0) comm.bcast(self, obj=None, int root=0)`

Collective Example

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
   data = {’key1’ : [7, 2.72, 2+3j],
           ’key2’ : ( ’abc’, ’xyz’)}
else:
   data = None
data = comm.bcast(data, root=0)
```

More Comprehensive mpi4py Tutorials

- basics - http://mpi4py.scipy.org/docs/usrman/tutorial.html
- advanced - http://www.bu.edu/pasi2011/01/Lisandro-Dalcin-mpi4py.pdf

4.14 Interesting Scalable Applications and Tools

- PyTrilinos - http://trilinos.sandia.gov/packages/pytrilinos/
- petsc4py - http://code.google.com/p/petsc4py/
- PyClaw - http://numerics.kaust.edu.sa/pyclaw/
- GPAW - https://wiki.fysik.dtu.dk/gpaw/

4.15 –> See Appendix 01 References for more resources