Note Taker Checklist Form -*MSRI*

Name : Rob Stapleton

E-mail Address/ Phone #: jrstaple@ncsu.edu

Talk Title and Workshop assigned to:
Interactive Parallel Computation in Support of Research in Algebra, Geometry, and Number Theory

Lecturer (Full name): Brian Granger

Date & Time of Event: 10:30 am Jan 28, 2007

Check List:

(✓) Introduce yourself to the lecturer prior to lecture. Tell them that you will be the note taker, and that you will need to make copies of their own notes, if any.

(✓) Obtain all presentation materials from lecturer (i.e. Power Point files, etc). This can be done either before the lecture is to begin or after the lecture; please make arrangements with the lecturer as to when you can do this.

(✓) Take down all notes from media provided (blackboard, overhead, etc.)

(✓) Gather all other lecture materials (i.e. Handouts, etc.)

(✓) Scan all materials on PDF scanner in 2nd floor lab (assistance can be provided by Computing Staff) – Scan this sheet first, then materials. In the subject heading, enter the name of the speaker and date of their talk.

Please do **NOT** use **pencil** or colored pens other than black when taking notes as the scanner has a difficult time scanning pencil and other colors.

Please fill in the following after the lecture is done:

1. List 6-12 lecture keywords: Python, IPython, parallel, interactive, SAGE, DSAGE, namespace

2. Please summarize the lecture in 5 or less sentences.
IPython is quickly becoming a major (the de facto) shell for scientific computing purposes. The goal is now to show off IPython's power when applied to parallel computing!

Once the materials on check list above are gathered, please scan ALL materials and send to the Computing Department. Return this form to Larry Patague, Head of Computing (rm 214)

10:30 a.m.
Interactive Parallel Computing with Python and IPython
Brian Granger

Python
-Freely available (BSD License)
-Highly portable: OSX, Windows, Linux, supercomputers
-Can be used interactively (like Matlab, Mathematica, IDL)
-Simple, expressive syntax readable by human beings
-Supports OO, functional, generic and meta programming
-Large community of scientific/HPC users
-Powerful built-in data types and libraries
    -Strings, lists, sets, dictionaries (hash tables)
    -Networking, XML parsing, thrading, regular expressions...
-Larger number of third party libraries for scientific computing
-Easy to wrap existing C/C++/Fortran codes

IPython: Enhanced Interactive Python Shell
-Freely available (BSD license) http://ipython.scipy.org/
-Goal: Provide an efficient environment for exploratory and interactive scientific computing
-The de facto shell for scientific computing in Python (has become in past few years)
-Available as a standard package on every major Linux distribution. Downloaded over 27k times in 2006 alone
-Interactive Shell fo rmany other projects:
    -Math (SAGE)
    -Astronomy (PyRAF,CASA)
    -Physics (Ganga, PyMAD)
    -Biology (Pymerase)
    -Web Frameworks

Capabilities:
-Input/output histories
-interactive GUI control: enables interactive plotting
-Highly customizable: extensible syntax, error handling, ...
-Interactive control system: magic commands
-Dynamic Introspection of nearly everything (objects, help, filesystem, etc.)
-Direc acess to filesystem and shell
-Integrated debugger and profiler support
-Easy to embed: give any program an interactive console with one line of code
-Interactive Parallel/Distributed Computing...

Traditional Parallel Computing: (at least in the realm of physics)
Compiled Languages:
-C/C++/Fortran are FAST for computers, SLOW for you
-Everything is low-level, you get nothing for free
    -Only primitive data types
    -Few built-in libraries
    -Manual memory management: bugs and more bugs
    -With C/C++ you don't even get built-in high performance numerical arrays

Message Passing Interface: MPI
Pros-
    -Robust, optimized, standardized, portable, common
    -Existing parallel libraries (FFTW, ScaLAPACK, Trillinos, PETSc)
    -Runs over Ethernet, Infiniband, Myrinet
    -Great at moving data around fast!

Cons-
	-Trivial things are not trivial. Lots of boilerplate code.
	-Orthogonal to how scientists think and work
	-Static: load balancing and fault tolerance are difficult to implement
	-Emphasis on compiled languages
	-Non-interactive and non-collaborative
	-Doesn't play well with other tools: GUIs, plotting, visualization, web
	-Labor intensive to learn and use properly

Case study: Parallel Jobs at NERSC in 2006:
NERSC = DOE supercomputing center at Lawrence Berkeley National Laboratory
Seaborg = IBM SP RS/6000 with 6080 CPUs
	-90% of jobs used less than 113 CPUs
	-Only 0.26% of jobs used more than 2048 CPUs
Jacquard = 712 CPU Operton system
	-50% of jobs used fewwer than 15 CPUs
	-Only 0.39% of jobs used more than 256 CPUs
Yes, this isn't the entire story. There are other issues involved, reasons people don't want to use lots of CPUs.

Realities:
-Developing highly parallel codes with these tools is extremely difficult and time consuming
-When it comes to parallel, WE are often the bottleneck
-We spend most of our time writing code rather than waiting for those "slow" computers
-With the advent of multi-core CPUs, this problem i scoming to a laptop/desktop near you
-Parallel speedups are not guaranteed!

Our Goals with IPython:
-Trivial parallel things should be trivial
-Difficult parallel things should be possible
-Make all stages of parallel computing fully interactive: development, debugging, testing, execution, monitoring, ...
-Make parallel computing more collaborative
-More dynamic model for load balancing and fualt tolerance.
-Seamless integration with other tools: plotting/visualization, system shell
-Also want to keep the benefits of traditional approaches:
	-Should be able to use MPI if it is appropriate
	-Should be easy to integrate compiled code and libraries.
-Support many types of parallelism.

Computing with Namespaces:
Namespaces:
-Namespace = a container for objects and their unique identifiers
	Very common with Python. Almost a hash table
-An instruction stream causes a namespace to evolve with time
-Interactive computing: the instruction stream has a human agent as its runtime source at some level
	Human is at the command at some point
-A (namespace, instruction stream) is a higher level abstraction than a process or a thread.
-Data in a namespace can be reated inplace (by instructions) or by external I/O (disk, network).

Important Points:
-Requirements for Interactive Computation:
	-Alice/Bob must be able to send instruction stream to a namespace
	-Alice/Bob must be able to push/pull objects to/from the namespace (disk, network)
-Requirements for Parallel Computation:
	-Multiple namespaces and instruction streams (for general MIMD parallelism).
	-Send data between namespaces (MPI is REALLY good at this).
-Requirements for Interactive Parallel Computation:
	-Alice/Bob must be able to send multiple instruction streams to multiple namespaces
	-User must be able to push/pull objects to/from the namespaces
THESE REQUIREMENTS HOLD for any type of parallelism

IPython Architecture:
pic

Architecture Details:
The IPython Engine/Controller/Client are typically different processes.
Why not threads? Later
-Can be run in arbitrary configurations on laptops, clusters, supercomputers
-Everything is asynchronous. Can't hack this on as an afterthought
-Must deal with long running commands that block all network traffic
-Dynamic process model. Engines and Clients can come and go at will at any time (unless you're using MPI)

Mapping Namespaces to Various Models of Parallel Computation:
Key Points:
-Most models of parallel/distributed computing can be mapped onto this architecture.
	-Message passing
	-Task farming
	-TupleSpaces
	-BSP (Bulk Synchronous Parallel)
	-Google's MapReduce
	-???
-With IPython's architecture of all these types of parallel computations can be done Interactively and collaboratively.
-The mapping of these models onto our architecture is done using interfaces + adapters and requires very little code.

Live Demo:
easy execution of commands on machines, easy to see what machines are connected, output of responses from machines as they
come in. One way they abstracte the asynchronicity is to use block=false to just return after it submits the command
Magic commands are simply handled by such example: %px import numpy
%px parallel execute
%pn # execute only on machine #
%result get results of last things
%autopx automatically send everything sent to all machines
rc.block=BOOL show blocking or not
rc.pushall Send data
rc.pullall Get data
Can use dictionary syntax: rc.pushAll(d=3564567) = rc['d'] = 3564567
rc.scatterAll Scatters an array across machines
rc.gatherAll Gathers the array back

The parallel time on the example rc.mapAll('lambda x: x**10', range (1000000)) is horrible
There's lots of tools here, but you still have to think.

Task-Based Computing:
-Common style of parallelism for loosely coupled or independent tasks
-more

Stepping Back a Bit:
-Event based systems are a nice alternative to threads:
    -Scale to multiple CPU systems
    -Bu9ild asynchronous nature of things in at low level
    -No deadlocks to worry about
-The networking framework used by IPython and SAGE (Twisted) has an abstraction (called a Deferred) for a result that will
arrive at some point in the future (like a promise in E)
-We have n interface that uses Deferreds to encapsulate asynchronous results/errors.

Benefits of an Event Based System:
-Arbitrary configurations of namespaces are immediately possible without worrying about deadlocks
    -Our test suite create a controller/client and multiple engines in a single process
-Possibiliities:
    -Hierarchies of client/contro...
    -Recursive systems

Error Propagation:
-Building a distributed system is easy...
-Unless you want to handle errors well
-We have spent a lot of time working/thinking about this issue. Not always obvious what should happen.
-Our goal: error handling/propagation in a parallel distributed context should be a nice analytic continuation of what
happens in a serial context.
-Remote exceptions should propagate to the user in a meaningful manner
-Policy of safety: don't ever let errors pass silently

This week:
-All the core developers of IPython's parallel capabilities are here at the workshop.

slides, black background → separate