Task-based programming with PyCOMPSs and its integration with data management activities at BSC

Clusters, Clouds, and Data for Scientific Computing
Châteauform, September 2-5 2014
Outline

- Programming challenges for the cloud and distributed computing in general
- StarSs programming model
- Programming with COMPSs
  - Python binding
  - COMPSs infrastructure and features
- Integration of COMPSs with new storage strategies
- Conclusions
Cloud programming challenge, or how to make it the programmers comfort zone

The Learning Zone model establishes a theory of how performance of a person can be enhanced and their skills optimized
- Comfort Zone: feel comfortable and do not have to take any risks
- Learning Zone: just outside of our secure environment, we grow and learn
- Panic Zone: all our energy is used up for managing/controlling our anxiety and no energy can flow into learning.

Moving to the learning zone, enables to extend the comfort zone, moving towards the panic zone

When following a personal dream or vision, individuals need to move to the learning zone and take controlled risks, in order to achieve the challenges of their panic zone

* The Learning Zone Model (Senninger, 2000)

Cloud poses different challenges to programmers
... away from the current comfort zone
... maybe in the panic zone???
The programming comfort zone

State of the art in programming
- Sequential programming
- Data is always where you expect
- All decisions controlled by the programmer

Programming for the cloud
- Parallel programming (~)
- Elasticity
- Distributed environment -> where is my data?
BSC vision

Applications

PM: High-level, clean, abstract interface

Power to the runtime

API

Program logic independent of computing platform

General purpose Task based Single address space

“Reuse” architectural ideas under new constraints
The StarSs programming model

- **StarSs**
  - Sequential general purpose programming language + annotations
  - Task based
  - Simple linear address space
  - Support for multiple platforms: SMP, GPUs, Cluster, Grids and Clouds

- **Programmability/Portability**
  - "Same" source code runs on "any" platform
  - Incremental parallelization/restructure
  - Focus in the problem, not in the hardware platform

- **Performance**
  - Intelligent Runtime
  - Automatically extracts and exploits parallelism
  - Locality awareness
  - Matches computations to specific resources on each type of target platform

Open Source
http://compss.bsc.es
The StarSs “Granularities”

**Average task Granularity:**
- 100 microseconds – 10 milliseconds
- 1 second - 1 day

**Address space to compute dependences:**
- Memory
- Files, Objects (SCM)

**Language binding:**
- C, C++, FORTRAN
- Java, Python, C/C++

**Parallel** | **Ensemble, workflow**
Advantages and drawbacks of COMPSs

✔ More flexible and with more expressivity
  – The potential of the programming language
  – Enables to express complex problems

✔ Data independent
  – Different data inputs may generate different task graphs

✔ Powerful runtime
  – Platform unaware
  – Exploits inherent parallelism

✗ Less explicit than graphical workflows
  – Although this can be partially compensated with the COMPSs monitor

✗ Large degree of flexibility may prevent some programmers to be efficient
  – Schemas such as MapReduce are sometimes more appreciated by programmers
  – Can be improved through training and support
Why Python?

Python is powerful... and fast; plays well with others; runs everywhere; is friendly & easy to learn; is Open. *

Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C.

Large community using it, including scientific and numeric.

Object-oriented programming and structured programming are fully supported.

Large number of software modules available (38,000 as of January 2014).

* From python.org
Python (PyCOMPSs) syntax

- Invoke tasks as Python functions/methods
- API for data synchronization

### Task selection in function definition (decorators)

**Function definition**

```python
@task(par = INOUT)
def myFunction(par):
...
```

```python
class Foo(object):
    @task()
def myMethod(self):
        ...
```

**Main Program**

```python
foo = Foo()
myFunction(foo)
foo.myMethod()
...
foo = compss_wait_on(foo)
foo.bar()
```
Custom Loader

initialize(f1);
for (int i = 0; i < 2; i++) {
    genRandom(f2);
    add(f1, f2);
}
print(f2);

User code: Python, Java, C/C++
Runtime System

- Task handling
  - Task graph
  - Scheduling
  - Data transfers
- Platform unawareness
  - Grid
  - Cloud
  - Cluster
- Cloud elasticity
- Support for heterogeneous clouds
  - i.e. openstack + Amazon
The BSC-CNS has been accredited with the Severo Ochoa Center of Excellence, an award given by the Spanish Ministry as recognition of leading research centres in Spain that are internationally known organisations in their respective areas.

Involves all BSC R&D departments

Four subprojects:
- Hardware and software technologies, to facilitate the introduction of Exascale computing and managing large amounts of data, focusing on the improvement of energy efficiency
- Personalized medicine, to design drugs to fit the needs of each patient
- Heart simulation, to perform modelling and simulation with the primary objective to determine how the heart muscle works
- Air quality and climate models, specially in areas that affect health (Sahara dust concentration)
Severo Ochoa: Cloud and BigData

- Intersection between Cloud Computing and large scale data analytics/management

- Vertical approach integrating previous technologies
  - Programming environments and runtime systems
  - Resource management in heterogeneous systems and workloads
  - Storage architecture and management

- To be demonstrated with “in-house” scientific challenges
**Human Brain Project**

- A 10-year European initiative to understand the human brain, enabling advances in neuroscience, medicine and future computing
- One of two FET Flagships
- A consortium of 256 researchers from 146 institutions, in 24 countries across Europe, in the US, Japan and China
- BSC contributes with programming models and resource management
Sample scenarios

Model = \{neurons\}

Simulation1

Simulation2

Analysis

Viz

Potentials = \{sequence for each neuron\}

Implementation: Persistent, Distributed, Resilient

API:
- Shared object space management: create/delete
- Access: get, put
- Query, iterators
- Concurrency
  - flow control (seq/par)
  - synchronization
- Consistency

Diagram:
- Model generator
- Simulation1
- Simulation2
- Analysis
- Viz
- API: Shared object space management
- Consistency
CS Software stack

Architectural design of Active Storage

COMPSs Apps

Common API (data access and control flow.)

Active Store

self-contained objects

Cassandra  PIMD BGAS  dataClay

Resource management policies:
data organization, query plans, computation scheduling

Hierarchical storage + Computing resources

Adaptive internal structure
Compute capability
**dataClay**: platform that manages **Self-Contained Objects** (data and code)

**Platform features:**
- Store and retrieve objects as seen by applications
- Remote execution of methods
- Add new classes
- Enrich existing classes: With new methods and With new fields
Bigdata resource management: overview

Objectives:
- to propose a highly-scalable resource management architecture for BigData applications
- to decouple data models from data organization
- to provide programmers with mechanisms to generate automatic data organization and automatic query code, that considers the performance of the data store system

Apache Cassandra used to evaluate our proposals
import sys

neurons_file_name = sys.argv[1]
correlation_file_name = sys.argv[2]

nd = NeuronData(neurons_file_name)
correlation = Correlation()

seed = 2398645
delta = 1782324

for i in nd.spikes.keys():
    for j in nd.spikes.keys():
        if i < j:
            cc_surrogate_range(i, j, nd,
            correlation, seed, num_surrs,
            num_bins,
            maxlag):
                sp = nd.spikes
                ...

                function definition

        seed = seed + delta

dumpToFile(correlation, corr_file_name)
import sys

neuron_data_name = sys.argv[1]
correlation_name = sys.argv[2]

nd = NeuronData(neuron_data_name)
correlation = Correlation()
correlation.make_persistent(correlation_name)

seed = 2398645
delta = 1782324

for block_i in nd.spikes.keys():
    for block_j in nd.spikes.keys():
        cc_surrogate_range(block_i, block_j, nd, correlation, seed, num_surrs, num_bins, maxlag):

sp = nd.spikes

for ni in block_i:
    for nj in block_j:
        if ni<nj:
            correlation.cc_originals[(ni,nj)] = correlate(sp[ni], sp[nj], ...)

my_cc_surrs[:,0] = surrs_ij_sorted[round(num_surrs*0.95),:]
my_cc_surrs[:,1] = surrs_ij_sorted[round(num_surrs*0.05),:]
correlation.cc_surrs[(ni,nj)] = my_cc_surrs
Neuroscience Data Processing @ PyCOMPSs and Cassandra: Data mapping

Programmer view

```python
class NeuronData:
    name = string
    spikes = dict

class Correlation:
    name = string
    cc_originals = dict
    cc_surrs = dict
```

Data identifier in persistent storage

Backend (Cassandra)

```
Name1 Table
key    spikes
key    value

Name2 Table
key    cc_originals    cc_surrs
key    value
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación
Conclusions

- Sequential programming approach
- Parallelization at task level
- Transparent data management and remote execution
- Can operate on different infrastructures: Cluster, Grid, Cloud (Public/Private)
- Enables orchestration of Web services
- Demonstrated in several projects and applications
- New language bindings (Python) and extensions to integrate with new storage methodologies make it a promising environment for Big-data projects
**COMPSs**

- Project page: http://www.bsc.es/compss
- Direct downloads page:
  - Source code
  - Sample applications & development virtual appliances
  - Tutorials
  - Red-Hat & Debian based installation packages
The COMPSs team

- Rosa M Badia
- Pedro Benedicte (part time)
- Carlos Diaz
- Jorge Ejarque
- Fredy Juarez
- Daniele Lezzi
- Francesc Lordan
- Roger Rafanell
- Cristian Ramon (part time)
- Raul Sirvent
- Enric Tejedor
Other CS members

- Toni Cortes
- Anna Queralt
- Jonathan Martí
- Jordi Torres
- Yolanda Becerra
- David Carrera
- Jesus Labarta
- Eduard Ayguadé
Thank you!

Downloads: http://www.bsc.es/computer-sciences/grid-computing/comp-superscalar/download
Support mailing list at http://compss.bsc.es/support-compss
Announces mailing list at http://compss.bsc.es/announces-compss
COMPSs Bindings

Java App

Python App

C/C++ App

Java Runtime

Python binding

C/C++ binding

C++ library

JNI
COMPSs Bindings: Objects (master)

- Primitive types mapped equivalent in Java
- Files also treated as files
- Objects are serialized to files by the Python binding
- Treated as files for the data-dependence
On task execution, serialized objects are sent as files to the worker.

The Python worker will deserialize the objects before invoking the task.
```python
import sys
from pycompss.api.api import compss_wait_on

num_frags = int(sys.argv[1])

# calculate number of pairs per fragment
num_ccs = (num_neurons**2 - num_neurons)/2
step = ceil(float(num_ccs)/num_frags)
start_idx = 0
end_idx = 0

seed = 2398645
delta = 1782324

@task(cc_original = INOUT, cc_surrs = INOUT)
def gather(result, cc_original, cc_surrs, start, end):
    cc_original[start:end, :] = result[0]
    cc_surrs[start:end, :, :] = result[1]

@task(returns = list)
def cc_surrogate_range(start_idx, end_idx, seed, num_neurons, num_surrs, num_bins, maxlag):
    ...

cc_original = zeros((num_ccs, 2*maxlag+1))
cc_surrs = zeros((num_ccs, 2*maxlag+1, 2))
for frag in range(num_frags):
    start_idx = end_idx
    end_idx = int(min((frag+1)*step, num_ccs))
    result = cc_surrogate_range(start_idx, end_idx, seed, num_neurons, num_surrs, num_bins, maxlag)
    gather(result, cc_original, cc_surrs, start_idx, end_idx)

seed = seed + delta

f = open('./result_cc_originals.dat', 'w')
cc_original = compss_wait_on(cc_original)
pickle.dump(cc_original, f)
f.close()

f = open('./result_cc_surrogates_conf.dat', 'w')
cc_surrs = compss_wait_on(cc_surrs)
pickle.dump(cc_surrs, f)
f.close()
```

import sys
num_frags = int(sys.argv[1])

# calculate number of pairs per fragment
num_ccs = (num_neurons**2 - num_neurons)/2
step = ceil(float(num_ccs)/num_frags)
start_idx = 0
end_idx = 0
seed = 2398645
delta = 1782324

for frag in range(num_frags):
    start_idx = end_idx
    end_idx = int(min((frag+1)*step, num_ccs))
    result = cc_surrogate_range(start_idx, end_idx, seed, num_neurons, num_surrs, num_bins, maxlag)
gather(result, originals_file, surrs_file)
    seed = seed + delta
Si poso totes les versions, sera massa codi.
Amb la de les taules de python per ilustrar la integració amb Cassandra seria suficient
STARS's basic idea

Sequential Code

for (i=0; i<N; i++) {
    T1 (data1, data2);
    T2 (data4, data5);
    T3 (data2, data5, data6);
    T4 (data7, data8);
    T5 (data6, data8, data9);
}

(a) Task selection + parameters direction (input, output, inout)

(b) Task graph creation based on data dependencies

(c) Scheduling, data transfer, task execution

(d) Task completion, synchronization

Parallel Computing Resources

Resource 1

Resource 2

... Resource N
Programming objectives

Reduce the development complexity of Grid/Cluster/Cloud applications to the minimum
- Writing an application for a computational distributed infrastructure may be as easy as writing a sequential application

Target applications: composed of tasks, called several times
- Granularity of the tasks or programs
- Data: files, objects, arrays and primitive types

Programming languages support
- Java (native)
- Python
- C/C++
Parallel python and workflows in Python

- Python threading module
  - Problem: global interpreter lock (GIL)
  - Is a mutex that prevents multiple native threads from executing Python bytecodes at once
  - CPython's memory management is not thread-safe
- Python multiprocessing module (and other fork-based solutions)
  - Bypass the GIL (multiple processes)
  - Explicit creation, synchronization, data sharing between processes
- MPI wrappers
  - mpi4py
  - pyMPI
- Libraries for embarrassingly-parallel distributed computations
  - ParallelPython - process-based, job-oriented solution with cluster support
  - dispy - Python module for distributing computations
  - … comprehensive list in: https://wiki.python.org/moin/ParallelProcessing
- Similarly, a quite comprehensive list of workflow modules
  - https://wiki.python.org/moin/FlowBasedProgramming
- PyCOMPSs
Neuroscience Data Processing example

Computation of mutual cross-correlations between all pairs of a set of spike data

Also computes the cross-correlations for surrogate data sets for each neuron pair

Sequential code

```python
f = open('./spikes.dat', 'r')
spikes = pickle.load(f)
f.close()

# preallocate result variables
num_ccs = (num_neurons**2 - num_neurons)/2
cc_orig = zeros((num_ccs,2*maxlag+1))
cc_surrs = zeros((num_ccs,2*maxlag+1,num_surrs))
idxrange = range(num_bins-maxlag,num_bins+maxlag+1)
row = 0

# for all pairs ni,nj such that nj > ni
for ni in range(num_neurons-1):
    for nj in range(ni+1,num_neurons):
        cc_orig[row,:] = correlate(spikes[ni,:],spikes[nj,:],
                                 num_spikes_i = sum(spikes[ni,:])
                                 num_spikes_j = sum(spikes[nj,:])
        for surrogate in range(num_surrs):
            surr_i = zeros(num_bins)
            surr_i[random.random_integers(0,num_bins-1,num_spikes_i)] = 1
            surr_j = zeros(num_bins)
            surr_j[random.random_integers(0,num_bins-1,num_spikes_j)] = 1
            cc_surrs[row,:,surrogate] = correlate(surr_i,surr_j,"full")[idxrange]
        row = row +1

# save results
f = open('./result_cc_originals.dat','w')
pickle.dump(cc_orig,f)
f.close()

f = open('./result_cc_surrogates.dat','w')
pickle.dump(cc_surrs,f)
f.close()
```
Main program

... # tuple of all parallel python servers to connect with
ppservers = ('comp1.my-network', 'comp2.my-network'
... 
if len(sys.argv) > 1:
    ncpus = int(sys.argv[1])
    # creates jobserver with ncpus workers
    job_server = pp.Server(ncpus, ...
else:
    # creates jobserver with workers automatically detected
    job_server = pp.Server(ppservers=ppservers, ...

# wait for servers to come up
time.sleep(5)

# calculate number of nodes in total
nlocalworkers = job_server.get_ncpus()
activenodes = job_server.get_active_nodes()
workerids = activenodes.keys()
nworkers = sum([activenodes[workerids[i]] for i in
range(len(workerids))]) + nlocalworkers
num_ccs = (num_neurons**2 - num_neurons)/2

# calculate number of pairs each worker should process
step = ceil(float(num_ccs)/nworkers)
start_idx = 0
end_idx = 0
starts = zeros((nworkers+1,))

Explicit resources declaration
Explicit fork join
Data back
Dump

Main program (cont)

for worker in range(nworkers):
    start_idx = end_idx
    end_idx = int(min((worker+1)*step,num_ccs))

    depfuncs = ()
    depmodules = "numpy","pickle",
    jobs.append(job_server.submit(cc_surrogate_range,...

    cc_original = zeros((num_ccs,2*maxlag+1))
    cc_surrs = zeros((num_ccs,2*maxlag+1))
    for worker in arange(nworkers):
        start = starts[worker]
        end = starts[worker + 1]
        result = jobs[worker]()
        cc_original[start:end,:] = result[0]
        cc_surrs[start:end,:,:] = result[1]

    f = open('./result_cc_originals.dat','w')
pickle.dump(cc_original,f)
f.close()

    f = open('./result_cc_surrogates_conf.dat','w')
pickle.dump(cc_surrs,f)
f.close()

Function definition

def cc_surrogate_range(start_idx, end_idx, seed, num_neurons,
num_surrs, num_bins, maxlag):
...
```python
import sys
from pycompss.api.api import compss_wait_on

num_frags = int(sys.argv[1])

# calculate number of pairs per fragment
num_ccs = (num_neurons**2 - num_neurons)/2
step = ceil(float(num_ccs)/num_frags)
start_idx = 0
end_idx = 0

seed = 2398645
delta = 1782324

cc_original = zeros((num_ccs,2*maxlag+1))
cc_surrs = zeros((num_ccs,2*maxlag+1,2))

for frag in range(num_frags):
    start_idx = end_idx
    end_idx = int(min((frag+1)*step,num_ccs))
    result = cc_surrogate_range(start_idx, end_idx, seed, num_neurons, num_surrs, num_bins, maxlag)
    gather(result, cc_original, cc_surrs, start_idx, end_idx)
    seed = seed + delta

f = open('./result_cc_originals.dat','w')
cc_original = compss_wait_on(cc_original)
pickle.dump(cc_original,f)
f.close()

f = open('./result_cc_surrogates_conf.dat','w')
cc_surrs = compss_wait_on(cc_surrs)
pickle.dump(cc_surrs,f)
f.close()
```
Neuroscience Data Processing: initial results

Execution time (MN III)

Task dependency graph

Execution trace
Testbed: minerva cluster
- 4 nodes, 16 cores each
- COMPSs → master running on one node, 4 cores
- Cassandra → virtual node configuration (60 = 12 + 16 + 16 + 16)
Initial tests:
- 100 keys (neurons)
- Random partitioning of keys over virtual nodes → 60 partitions
- 3600 tasks, one for each pair of blocks (60 x 60)
- Unbalance due to:
  - Key distribution
  - Each task computes a different number of correlations
Improved balancing of keys in Cassandra
- Manual assignment of token ranges to vnodes

Results (100 keys)
- Blocks with uniform number of keys lead to better balancing
- The algorithm itself still causes unbalance
  - Distributed work stealing could help with this