Parallelizing Python applications with PyCOMPSs

Rosa M Badia,

Intel® HPC Developer Conference 2017
Outline

• Welcome and introduction (5 min)
• Section 1: Introduction to PyCOMPSs (25 mins)
  • Programming model
  • Python syntax
  • First Python example
• Section 2: Examples and performance results (15 mins)
  • Linear algebra + Intel® MKL
  • Kmeans
• Section 3: Hands-on with Jupyter notebook (30 mins)
  • Initial demos: execution and monitoring
  • Set-up with Google
  • Execution in Google environment with Jupyter notebook
    • K-Means
    • Cholesky
• Questions and answers (10 mins)
• Conclusions (5 mins)
• Backup material
  • Section 4: COMPSs runtime
    • Runtime components
    • Execution environments
    • Interaction with Intel® MKL
Section 1: Introduction to PyCOMPSs
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 (+115,000 as of August 2017 **)
Programming with PyCOMPSs

- Sequential programming
- General purpose programming language + annotations/hints
  - To identify tasks and directionality of data
- Task based: task is the unit of work
- Simple linear address space
- Builds a task graph at runtime that express potential concurrency
  - Implicit workflow
- Exploitation of parallelism
  - ... and of distant parallelism
- Agnostic of computing platform
  - Enabled by the runtime for clusters, clouds and grids
PyCOMPSs runtime

- Sequential execution starts in master node
- Tasks are offloaded to worker nodes
- All data scheduling decisions and data transfers performed by runtime
PyCOMPSs Syntax

• Python decorators:

```python
from pycompss.api.task import task

def function():
    # do something

@task
```

The decorator is used to indicate that the function is considered a task definition.

Each call to the function will be considered as a task call.

• API:

```python
if __name__ == '__main__':
    from pycompss.api.api import compss_wait_on
    # do some task calls
    compss_wait_on(something)
```

The API is used to indicate that a synchronization is requested.
PyCOMPSs: Task definition

- Task definition with Python decorators
  - Provide information about task parameters (TYPE_DIRECTION):
    - Type
      - Only mandatory for files
      - Inferred for the rest of the types
    - Direction
      - Default IN (read-only)
      - Mandatory for INOUT (read-write) and OUT (write-only)

```python
@task(a = INOUT, b = FILE_OUT)
def my_func(a, b, c):
    ...
```

- Type inferred
- Explicit type and direction
- Type inferred, default direction (IN)
PyCOMPSs: Task definition (II)

• The @task decorator: special arguments

  • Type of the return value → mandatory if a value is returned

```python
@task(returns = int)
def ret_func():
    return 1
```

  • The function may return more than one value:

```python
@task(returns = (int, list))
def ret_func():
    return 1, [2, 3]
```
PyCOMPSs: Task definition (III)

• The @task decorator: special arguments

  • Does the task modify the callee object? → default True

    ```python
    class MyClass(object):
        @task(isModifier = False)
        def instance_method(self):
            ...# self is NOT modified here
    ```

  • Is it a priority task? → Default False

    ```python
    @task(priority = True)
    def prio_func():
        ...
    ```
PyCOMPSs: Synchronization API

• Data created or updated by a task can be used in the main program of the application
  • But we need to synchronize first!

• Three API methods for synchronization

```python
my_file = 'file.txt'
func(my_file)
fd = compss_open(my_file)
...

my_obj = MyClass()
my_obj.method()
my_obj = compss_wait_on(my_obj)
...

... compss_barrier() ...
```

*func* is a task that modifies *my_file*

*method* is a task that modifies *my_obj*

Barrier - does not synchronize data useful for measuring time
PyCOMPSs: dependency detection

- Dependencies created both from function parameters (OUT, INOUT) and return values
- Return value is equivalent to OUT

```python
@task(a = INOUT, b = FILE_OUT)
def func_1(a, b, c):
    ...

@task()
def func_2(g):
    return MyClass(...)

@task()
def func_3(d, e, f):
    ...
```

```python
x=5
func_1 (x, y, z)
r= func_2 (x)
func_3 (r, s, p)
...
```
Why a synchronization call?

• Data dependencies are detected between tasks’ arguments of functions and results
• However, what happen if code outside tasks want to access results (return value or output arguments) of tasks?
  • Since tasks are executed asynchronously, there is no evidence that the result have been generated
  • Since PyCOMPSs executes in a distributed computing platform, the results may reside in a remote storage
  • Then is when the synchronization calls should be used

```python
@task(a = INOUT, b = FILE_OUT)
def func_1(a, b, c):
    ...

@task(returns = MyClass)
def func_2(g):
    return MyClass(...)  # OUT

@task()
def func_3(d, e, f):
    ...

x=5
func_1 (x, y, z)
r = func_2 (x)
r = compss_wait_on (r)
r = r+1
...```

IN
PyCOMPSs: Constraints

• Enables definition of tasks’ constraints
  • Resource to execute the task should meet the constraint

• Decorator definition:
  • `@constraint(constraint1="value1", constraint2="value2", ...)`

• Examples of supported constraints:
  • ProcessorArchitecture
  • ComputingUnits
  • MemorySize
  • AppSoftware

```python
@constraint(ComputingUnits="8")
@task(A=INOUT, priority=True)
def potrf(A):
    A.dpotrft(lower=True)
```
Support for tasks’ polymorphism

• Mechanism to support multiple implementations of a single behavior
• Combined with constraints to select different architectures
• Runtime selects more adequate version – depends on actual scheduling policy

```python
@implement (source class="matmul objects MKL", method="multiply")
@constraint (ComputingUnits="$ComputingUnitsKNL", ProcessorName="KNL")
@task (c=INOUT)
def multiplyKNL(a, b, c, MKLProcXeon, MKLProcKNL):
    os.environ["KMP AFFINITY"]="disabled"
    os.environ["MKL NUM THREADS"]=str(MKLProcKNL)
    c += a * b
```

```python
@constraint (ComputingUnits="$ComputingUnitsXEON", ProcessorName="XEON")
@task (c=INOUT)
def multiply(a, b, c, MKLProcXeon, MKLProcKNL):
    os.environ["MKL NUM THREADS"]=str(MKLProcXeon)
    c += a * b
```
PyCOMPSs: other decorators

• Definition of specific task implementations:
  • Binary
  • MPI
  • OmpSs

• Decorator definition:
  • @binary(binary=“app.bin”)
  • @mpi(binary=“mpiApp.bin”, runner=“mpirun”, computingNodes=8)
  • @ompss(binary=“ompssApp.bin”)

• Can be combined with the @constraint and @implement decorators

```python
@implement... (optional)
@constraint... (optional)
@binary (binary=“app.bin”, workingDir=“/myApp”)
@task()
def func(l):
    pass
```
Sample code

- PyCOMPSs word-count
- Tasks operate on blocks of input data
- Task-graph concurrency can sometimes be changed to improve concurrency:
  - Calling the reduce tasks to create a tree

Tasks definition

```python
@task(returns=dict)
def word_count(collection):
  ...

@task(dict_1=INOUT)
def reduce_count(dict_1, dict_2):
  ...
```

Main Program

Data = [block1, block2, …, blockN]
result = defaultdict(int)
for block in Data:
    presult = word_count(block)
    reduce_count(result, presult)
finalResult = compss_wait_on(result)

Dataflow:

1. Data = [block1, block2, …, blockN]
2. result = defaultdict(int)
3. for block in Data:
   3.1. presult = word_count(block)
   3.2. reduce_count(result, presult)
4. finalResult = compss_wait_on(result)
Section 2: Examples and performance results
Matrix multiply

• Blocked matrix
• Tasks operates on blocks of the matrices
• Blocks are NumPy arrays and operate on NumPy operations

@constraint (ComputingUnits="$ComputingUnits")
@task (c=INOUT)
def multiply(a, b, c, MKLProc):
    os.environ["MKL_NUM_THREADS"] = str(MKLProc)
    c += a * b

startTime = time.time()  
initialize_variables()  
compss_barrier()  
initTime = time.time() - startTime92
startMulTime = time.time()  
for i in range(MSIZE):
    for j in range(MSIZE):
        for k in range(MSIZE):
            multiply (A[i][k], B[k][j], C[i][j], MKLProc)  
compss_barrier()  
mulTime = time.time() - startMulTime  
totalTime = time.time() - startTime
Matrix multiply

- Initialization performed with tasks
  - Memory is allocated in the node where the task is executed
- Scheduling will try to place tasks using the same blocks in the same node, reducing transfers and other overheads

```python
@constraint(ComputingUnits="$ComputingUnits")
@task(returns=list)
def createBlock(BSIZE, res, MKLProc):
    os.environ["MKL_NUM_THREADS"] = str(MKLProc)
    if res:
        block = np.array(np.zeros((BSIZE, BSIZE)), dtype=np.double, copy=False)
    else:
        block = np.array(np.random.random((BSIZE, BSIZE)), dtype=np.double, copy=False)
    mb = np.matrix(block, dtype=np.double, copy=False)
    return mb

def initialize_variables(MKLProc):
    for matrix in [A, B]:
        for i in range(MSIZE):
            matrix.append([])
        for j in range(MSIZE):
            mb = createBlock(BSIZE, False, MKLProc)
            matrix[i].append(mb)
    for i in range(MSIZE):
        C.append([])
        for j in range(MSIZE):
            mb = createBlock(BSIZE, True, MKLProc)
            C[i].append(mb)
```
Cholesky factorization

- Cholesky factorization, right-looking approach
- Blocked algorithm
- Functions in bold annotated as tasks

```python
def cholesky_blocked(A):
    import os
    for k in range(MSIZE):
        # Diagonal block factorization
        A[k][k] = potrf(A[k][k])
        # Triangular systems
        for i in range(k+1, MSIZE):
            A[i][k] = solve_triangular(A[k][k], A[i][k])
            A[k][i] = np.zeros((BSIZE, BSIZE))
        # update trailing matrix
        for i in range(k+1, MSIZE):
            for j in range(i, MSIZE):
                A[j][i] = gemm(-1.0, A[j][k], A[i][k], A[j][i], 1.0)
    return A
```

main
Cholesky factorization

```python
@constraint(ComputingUnits="${ComputingUnitsXEON}", ProcessorName="XEON")
def potrf(A):
    from scipy.linalg.lapack import dpotrf
    import os
    os.environ['MKL_NUM_THREADS']=str(mkl_threads)
    A = dpotrf(A, lower=True)[0]
    return A

@constraint(ComputingUnits="${ComputingUnitsXEON}", ProcessorName="XEON")
def solve_triangular(A, B):
    from scipy.linalg import solve_triangular
    from numpy import transpose
    import os
    os.environ['MKL_NUM_THREADS']=str(mkl_threads)
    B = transpose(B)
    B = solve_triangular(A, B, lower=True)  #, trans='T'
    B = transpose(B)
    return B
```
Cholesky factorization

Task versions

```python
@constraint(ComputingUnits="\${ComputingUnitsXEON}\", ProcessorName="XEON")
@task(returns=list)
def gemm(alpha, A, B, C, beta):
    from scipy.linalg.blas import dgemm
    from numpy import transpose
    set_mkl_threads(MKLProcXeon)
    B = transpose(B)
    C = dgemm(alpha, A, B, c=C, beta=beta)
    return C

@implement(source_class="cholesky", method="gemm")
@constraint(ComputingUnits="\${ComputingUnitsKNL}\", ProcessorName="KNL")
@task(returns=list)
def gemm_knl(alpha, A, B, C, beta):
    from scipy.linalg.blas import dgemm
    from numpy import transpose
    set_mkl_threads(MKLProcKNL)
    B = transpose(B)
    C = dgemm(alpha, A, B, c=C, beta=beta)
    return C
```
Other examples

• Cholesky – implementation based on right looking approach
  • Task graph for a 16 x 16 blocks factorization
Matmul evaluation

- Sequential annotated code runs in MareNostrum III, distributed through several nodes
  - 2x Intel® SandyBridge-EP E5-2670, 8 cores/socket
  - 32/128 GB per node
  - Infiniband network

Matrix size: 64K x 64 K doubles
Block size: 4096x4096 each block
4 tasks per node
16 Intel® MKL threads / task
oversubscription = 4

Baseline: PyCOMPSs execution with one worker node (whole matrix)
Matmul evaluation

• Results in SSF cluster, distributed through several nodes
  • Lustre as shared file system
  • 2 x Intel® Xeon™ processor E52690 – V4, 14 cores/socket, 2-threads/core
  • 110 GB/node

Matrix size: 128K x 128K doubles
Block size: 8192x8192 each block
8 tasks per node
7 Intel® MKL threads / task
No oversubscription
Master in first worker node

Baseline: Maximum performance
Obtained with multi-threaded Intel® MKL in a single node
with a block of size 16384 x 16384
Matmul evaluation - heterogeneous nodes

- Results in SSF cluster, distributed through several heterogeneous nodes
  - Intel® Xeon™ processor E52690 – V4 (Xeon™ nodes)
    - 2 sockets/node, 14 cores/socket, 2-threads/core
  - Intel® Xeon Phi™ 7210 (KNL nodes)
    - 64 cores, 4 threads per core
- Performance using different schedulers

Matrix = 131K X 131K doubles
Xeon™ nodes: 8 tasks/node, 7 threads/task
KNL nodes: 4 tasks/node, 64 threads/task
Intel® Distribution for Python 2.7.13
Block size: 4096 x 4096
Cholesky evaluation

• Results in SSF cluster, distributed through several nodes
  • Intel® Xeon™ E52690 – V4 with lustre as shared file system
  • 2 sockets/node
  • 14 cores/socket, 2-threads/core

• Evaluation of different block sizes

Matrix = 128K X 128K doubles
8 tasks per node
7 threads per task (oversubscription = 1)
Intel® Distribution for Python 2.7.13

Baseline: Maximum performance obtained with multi-threaded Intel® MKL in a single node for a block of size 8192 x 8192
Cholesky evaluation—heterogeneous nodes

- Results in SSF cluster, distributed through several heterogeneous nodes
  - Intel® Xeon™ E52690 – V4 (Xeon™ nodes)
    - 2 sockets/node, 14 cores/socket, 2-threads/core
  - Intel® Xeon Phi™ 7210 (KNL nodes)
    - 64 cores, 4 threads per core

Matrix = 128K X 128K doubles
Xeon™ nodes: 8 tasks/node, 7 threads/task
KNL nodes: 4 tasks/node, 64 threads/task
Intel® Distribution for Python 2.7.13
Block size: 4096 x 4096
Cholesky evaluation—heterogeneous nodes

- Timeline of tasks
- Intel® Xeon Phi™ (KNL) nodes only execute dgemms
  - Selected with constraints decorator
K-means evaluation

6.400,000 points
128 dimensions
100 centers
10 iterations
Fragments: 4 fragments / CPU
56 tasks per node
1 MKL threads / task
Master in first worker node

- Results in SSF cluster, distributed through several nodes
  - Lustre as shared file system
  - 2 x Intel Xeon™ E52690 – V4, 14 cores/socket, 2-threads/core
  - 110 GB/node

Elapsed time

Speed-up

- Results in SSF cluster, distributed through several nodes
  - Lustre as shared file system
  - 2 x Intel Xeon™ E52690 – V4, 14 cores/socket, 2-threads/core
  - 110 GB/node

Elapsed time

Speed-up
Section 3: Hands-on with Jupyter notebook
The Jupyter Notebook is a web application that allows you to create and share documents that contain live code, equations, visualizations and explanatory text.

Uses include: data cleaning and transformation, numerical simulation, statistical modeling, machine learning and much more.

Runs Python – sequential

Prototype of PyCOMPSs integrated with Jupyter notebook

- Runs in parallel in local node and can offload tasks to external nodes
PyCOMPSs @ Jupyter notebook

- Runtime started from notebook
- PyCOMPSs tasks registered and send to workers
- Apps can be configured to generate trace, graph and to be monitored
Hands-on outline

• Hands-on with PyCOMPSs in Jupyter notebook
• Virtual image with PyCOMPSs and other tools installed in google cloud services
• Small usage demo
• Hands-on with K-means example
  • Code exploration
  • Execution in Jupyter notebook
  • Monitoring execution with COMPSs monitor
• Repeat steps with Choleski factorization code
PyCOMPSs @ Jupyter notebook

• Small usage demo
PyCOMPSs and Jupyter-Notebook @ Google cloud

• For Linux and Mac OS: start two sessions in your virtual instance, eg:

```
ssh -i ./Documents/Certificats/rosa.pem rosa@104.198.40.18
```

Use your user-id and IP of your instance

• In case using MS Windows, open two consoles with PuTTY or another ssh client
PyCOMPSs and Jupyter-Notebook @ Google cloud

• In one of the consoles, first time only:
  • Change to tutorial_apps folder:
    > cd tutorial_apps
  • Run init_keys.sh script to generate SSH keys
    > ./init_keys.sh
  • Start the COMPSs monitor
    > ./init_monitor.sh

• In one of the consoles (every time):
  • Start Jupyter Notebooks without browser:
    > jupyter-notebook --ip=0.0.0.0 --no-browser

• From your favorite browser:
  • Open the Jupyter-notebook client, using the token given in the console, eg:
    http://104.198.40.18:8888/?token=c154f3b14a24827b2273732a4bc27537ec24271c9e33f27d
  • In another tab, open the compss-monitor:
    http://104.198.40.18:8080/compss-monitor/zul/login.zul
  • Login with your user-id (eg: rosa)

use your machine IP
PyCOMPSs: Kmeans @ Jupyter

- Kmeans hands-on with Jupyter notebook
- Navigate to folder /tutorial_apps/python/jupyterhands-on
PyCOMPSs Application – Kmeans (I)

• Initial steps:

```python
In [ ]: import pycompss.interactive as ipycompss

In [ ]: ipycompss.start(graph=True, monitor=2000)
    #trace=True

In [ ]: from pycompss.api.task import task

In [ ]: import numpy as np
```

• Data initialization

```python
In [ ]: def init_random(numV, dim, seed):
    np.random.seed(seed)
    c = [np.random.uniform(-3.5, 3.5, dim)]
    while True:
        yield np.random.uniform(-3.5, 3.5, dim)
    return

In [ ]: # Not a task for plotting
    def genFragment(numV, K, c, dim, mode='gauss'):
        if mode == "gauss":
            n = int(float(numV) / K)
            r = numV % K
            data = []
            for k in range(K):
                s = np.random.uniform(0.05, 0.75)
                for i in range(n+r):
                    d = np.array([np.random.normal(c[k][j], s) for j in range(dim)]
                    data.append(d)
                return np.array(data)[:numV]
        return [np.random.random(dim) for _ in range(numV)]
```
PyCOMPSs Application – Kmeans (II)

• Tasks’ code:

```python
In [ ]: @task(returns=dict)
def cluster_points_partial(XP, mu, ind):
dic = {}
    for x in enumerate(XP):
        bestmukey = min(((i[0], np.linalg.norm(x[1] - mu[i[0]])) for i in enumerate(mu)), key=1:)
        if bestmukey not in dic:
            dic[bestmukey] = [x[0] + ind]
        else:
            dic[bestmukey].append(x[0] + ind)
    return dic
```

```python
In [ ]: @task(returns=dict)
def partial_sum(XP, clusters, ind):
dic = {}
    for i, l in p:
        dic[i] = (len(l), np.sum(l, axis=0))
    return dic
```
PyCOMPSs Application – Kmeans (III)

- Reduction task and algorithmic scheme for tree-reduction

```python
In [ ]: def reduceCentersTask(a, b):
    for key in b:
        if key not in a:
            a[key] = b[key]
        else:
            a[key] = (a[key][0] + b[key][0], a[key][1] + b[key][1])
    return a

In [ ]: def mergeReduce(function, data):
    from collections import deque
    q = deque(range(len(data)))
    while len(q):
        x = q.popleft()
        if len(q):
            y = q.popleft()
            data[x] = function(data[x], data[y])
            q.append(x)
        else:
            return data[x]
```
PyCOMPSs Application – Kmeans (IV)

- Main code

```python
In [ ]:

```from pycompss.api.api import compss_wait_on

```%matplotlib inline

```numV = 1000000  # Vectors ------ with 1000 it is feasible to see the evolution across iterations
dim = 2  # Dimensions
k = 4  # Centers
numFrag = 16  # Fragments
epsilon = 1e-10  # Convergence condition
maxIterations = 20  # Max number of iterations

size = int(numV / numFrag)
cloudCenters = init_random(k, dim, 8) # centers to create data groups
X = [genFragment(size, k, cloudCenters, dim, mode='gauss') for _ in range(numFrag)]

mu = init_random(k, dim, 7) # First centers
oldmu = []
n = 0

while not has_converged(mu, oldmu, epsilon, n, maxIterations):
    oldmu = mu
    clusters = [cluster_points_partial(X[f], mu, f * size) for f in range(numFrag)]
    partialResult = [partial_sum(X[f], clusters[f], f * size) for f in range(numFrag)]
    mu = mergeReduce(reduceCentersTask, partialResult)
    mu = compss_wait_on(mu)
    mu = [mu[c][1] / mu[c][0] for c in mu]
    n += 1

clusters = compss_wait_on(clusters)
plotKMEANS(dim, mu, clusters, X)

print "Result:"
print "Iterations: ", n
print "Centers: ", mu
```
PyCOMPSs Application – Kmeans (V)

• Runtime finalization:

```
In [ ]: ipycompss.stop()
```

• If anything goes wrong, you can stop all COMPSs processes with the clean_all_compss.sh script:

```
> ./clean_all_compss.sh
```
Final notes
Installation

- Installation manual:

- Source code:
  - http://compss.bsc.es/ (Downloads Section – Source)

- Packages and repositories:
  - http://compss.bsc.es/ (Downloads Section – Repository references)
    - Debian based: apt-get install compss-framework
    - Zypper based: zypper install compss-framework
    - Yum based: yum install compss-framework

- Supercomputers:
  - $ wget http://compss.bsc.es/repo/sc/stable/COMPSs_2.0.tar.gz
  - $ tar -xvzf COMPSs_2.0.tar.gz
  - $ cd COMPSs
  - $ ./install <targetDir>

- Pip:
  - sudo -E pip install compss --v
  - source /etc/profile.d/compss.sh
Final Notes

• Sequential programming approach
• Parallelization at task level
• Transparent data management and remote execution

• Can operate on different infrastructures:
  • Cluster, Grid, Cloud (Public/Private)
  • Also with containers’ technology: Docker, Singularity, Mesos

• Other PyCOMPSs/COMPSs events at SC17:
  • “Enabling Python to execute efficiently in heterogeneous distributed infrastructures with PyCOMPSs”, Sunday 12\textsuperscript{th} Nov – PyHPC workshop
  • “Enabling GPU support for the COMPSs-Mobile framework”, Monday 13\textsuperscript{th} Nov – WACCPD workshop
  • Demos at the BSC booth, no. 1975
Final Notes

• Project page:
  • http://www.bsc.es/compss

• Direct downloads page:
  • http://www.bsc.es/computer-sciences/grid-computing/comp-superscalar/download
    • Virtual Appliance for testing & sample applications
    • Tutorials
    • Red-Hat & Debian based installation packages
    • Source Code

• Application Repository
  • http://compss.bsc.es/projects/bar/wiki/Applications
    • Several examples of applications developed with COMPSs
Workflows and distributed computing group (COMPSs)

- Adrià Aguilar
- Javier Alvarez
- Pol Alvarez
- Rosa M Badia
- Arnau Canyadell
- Javi Conejero
- Jorge Ejarque
- Daniele Lezzi
- Francesc Lordan
- Cristian Ramon-Cortés
- Sergio Rodriguez
- Carlos Sagarra
- Albert Serven
Projects where COMPSs is used/developed

- Multiscale Complex Genomics
- mFAC
- TANGO
- bioexcel
- EUBRA BIGSEA
- nextgenio
- Human Brain Project

Intel® HPC Developer Conference 2017
THANK YOU!

For further information please contact support-compss@bsc.es

www.bsc.es
Section 2: COMPSs runtime
**COMPSs Overview - Runtime System**

- Componentized
- Adaptable
- Extensible
- Each component responsible of a specific task
- Task generation
  - Dependence analysis
  - Generation of task graph
- Task scheduling
  - When?
  - To which resource?
- Data management
  - Where is the data?
  - Transfer of data
COMPSs Overview - Runtime System

• Access processor
  • Performs data management
  • Knows where data is

• Task dispatcher
  • Schedules tasks
  • Finds dependencies
  • Adds tasks to task-graph
  • Updates task-graph

• Monitor executor
  • Monitors execution at real-time

• Resource optimizer
  • Decides on creation of new machines
  • Cloud only

Intel® HPC Developer Conference 2017
Runtime System

Application

Task Selection Interface

How do I select the execution platform?

Grid

Cluster

Cloud
COMPSs Execution Environment

Infrastructure Description
• Describes the available resources in the infrastructure
• Describes Cloud Providers: Images and VM Templates

Adaptors
- Network adaptors
  - NIO
  - GAT

Resources
- Cloud connectors
  - jClouds
  - rOCCI

resources.xml
project.xml
COMPSs Execution Environment

Adaptors
- Network adaptors
  - NIO
  - GAT

Resources
- Cloud connectors
  - jClouds
  - rOCCI

Application Execution Description
- Selection of resources
- Application Code Location
- Working directory

project.xml

resources.xml
COMPSs Execution Environment

Master-Worker Communication Mechanism
- GAT: Restricted environments (only ssh access) and Grid Middleware
- NIO: Efficient Persistent workers implementation
- Controlled and secured environments

Adaptors
- Network adaptors
  - NIO
  - GAT

Resources
- Cloud connectors
  - jClouds
  - rOCCI

Resources
- project.xml
- resources.xml
COMPSs Execution Environment

Resource Scalability
- jClouds: access to most of commercial public clouds
- rOCCI: OGF standard
- Extensible (support others..)

Adaptors
- Network adaptors
  - NIO
  - GAT

Resources
- Cloud connectors
  - jClouds
  - rOCCI

resources.xml
project.xml
PyCOMPSs in remote hosts (interactive)

- Typical setup:
  - Master node: main program (+ master runtime)
  - Worker nodes: tasks (+ worker runtime)
- PyCOMPSs executions submitted with `runcompss` command
PyCOMPSs in a Cluster (with job scheduler)

- Execution divided in two phases
  - Job-submission of a whole COMPSs app execution – *enqueue_compss*
    - Project.xml and Resource.xml generated automatically
  - Application execution when allocation is obtained

![Diagram showing PyCOMPSs in a Cluster (with job scheduler)]
COMPSs in Clouds

• Execution of COMPSs applications in Clouds
  • Select de connector to interact with the Cloud provider
  • Adaptor to communicate VMs (NIO if provider supports firewall management, GAT if only ssh)
  • Execution with runcompss command
  • Support for elasticity
Elasticity with SLURM

1. `sbatch --dependency=expand:X -N1 -n.. --mem=..` (Requests a new node)

2a. `squeue -j Y` (check job Y status)

2b. SLURM creates the new job

3. `scontrol update JobId=X NumNodes=ALL ...` (update original job properties)
PyCOMPSs with Docker

- Keep as transparent for the user as possible
- Same as running a remote COMPSs application (runcompss command)
- Deploy applications as a set of docker container

runcompss_docker
PyCOMPSs stack + Numpy and MKL

- Use of two level parallelism: task level and thread level (MKL)
- PyCOMPSs runtime binds MKL threads to single socket
- Programmer responsible of defining block size and threads to be used