Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

Task-based parallel programming model in Python

Rosa M Badia

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Intel-BSC Exascale Lab

Collaboration between Intel and the Barcelona Supercomputing Center

10-13/09/2017

PPAM 2017, Lublin

The MareNostrum 4 Supercomputer

In operation since July 2017

Total peak performance

13.7 Pflops/s, 390TB

12 times more powerful than MareNostrum 3

Compute

General Purpose, for current BSC workload

More than 11 Pflops/s

3,456 nodes of Intel Xeon v5 processors

Emerging Technologies, for evaluation of 2020 Exascale systems

3 systems, each of more than 0.5 Pflops/s with KNL/KNH, Power9+NVIDIA, ARMv8

Storage

14 PB of GPFS

Storage System

Network

IB EDR/OPA

Ethernet

Operating System: SuSE

13th in TOP500

3rd in Europe

80% PRACE users

16% Spanish researchers

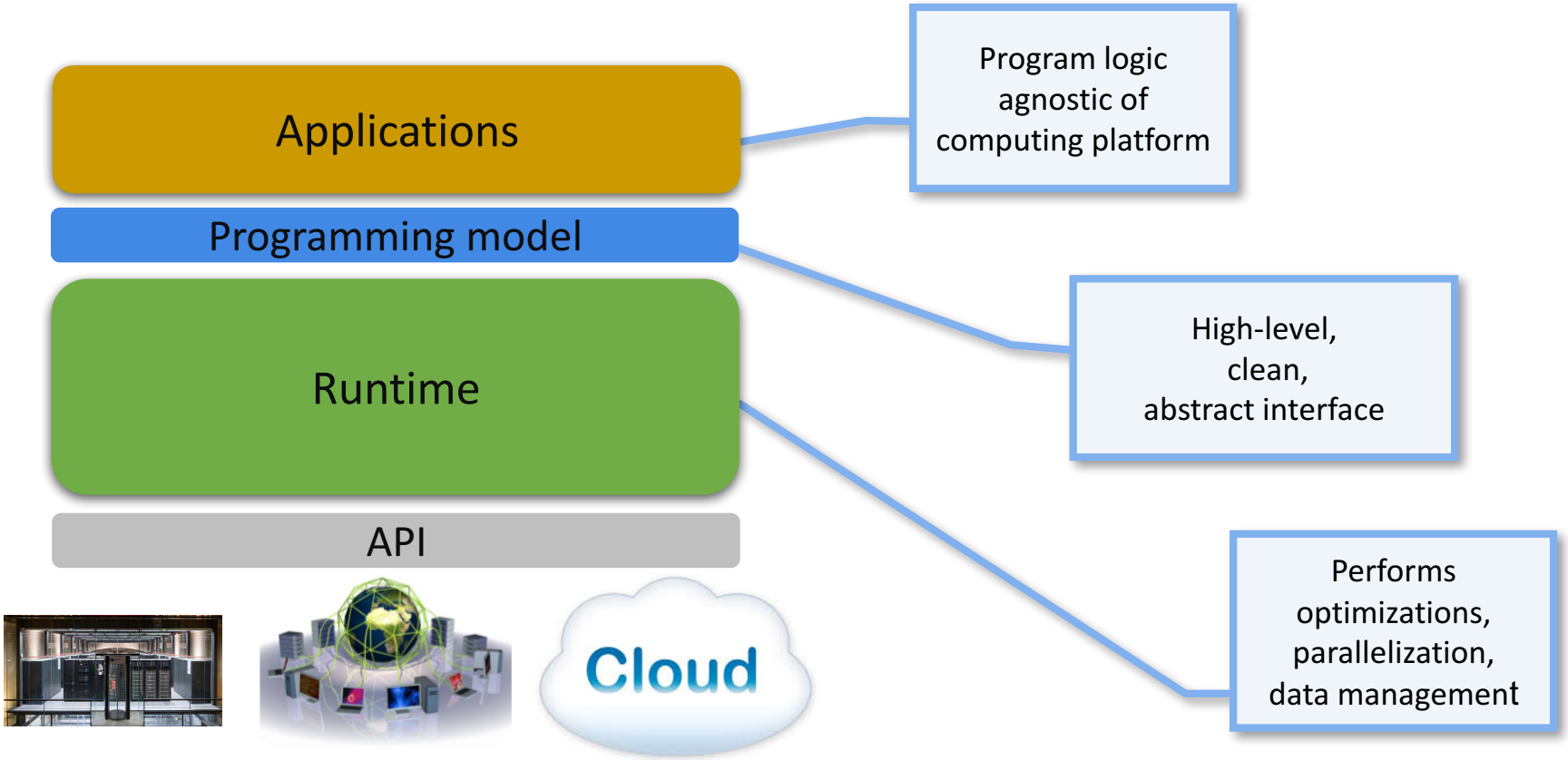
4% BSC researchers

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Challenges

- New architectures and organization of processors
  - Multicore
    - Including vector units
  - GPU/accelerators
  - FPGAs
- Shift on programming paradigms:
  - From sequential to parallel
  - New instructions/languages
- Storage systems
  - New storage technologies
  - Shift on storage paradigm

BSC vision on programming models



Outline

- PyCOMPSs overview
- Integration with new storage technologies
- Applying to linear algebra kernels
- Performance results in several architectures
- Conclusions

www.bsc.es

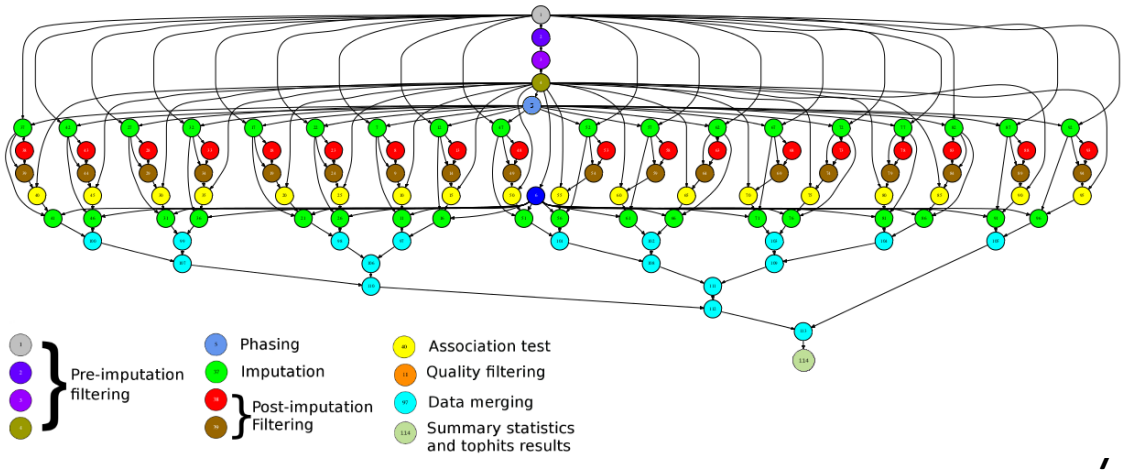
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

PyCOMPSs/COMPSs overview

Programming with PyCOMPSs/COMPSs

- 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

- Based on regular/sequential Python code
- Use of decorators to annotate tasks and indicate arguments directionality
- Other annotations: constraints
- Small API for data synchronization

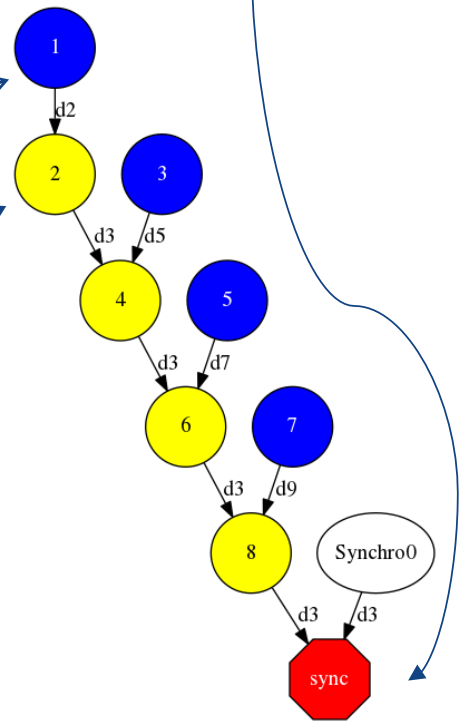
Main Program

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

Tasks definition

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

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



Tasks' contraints and versions

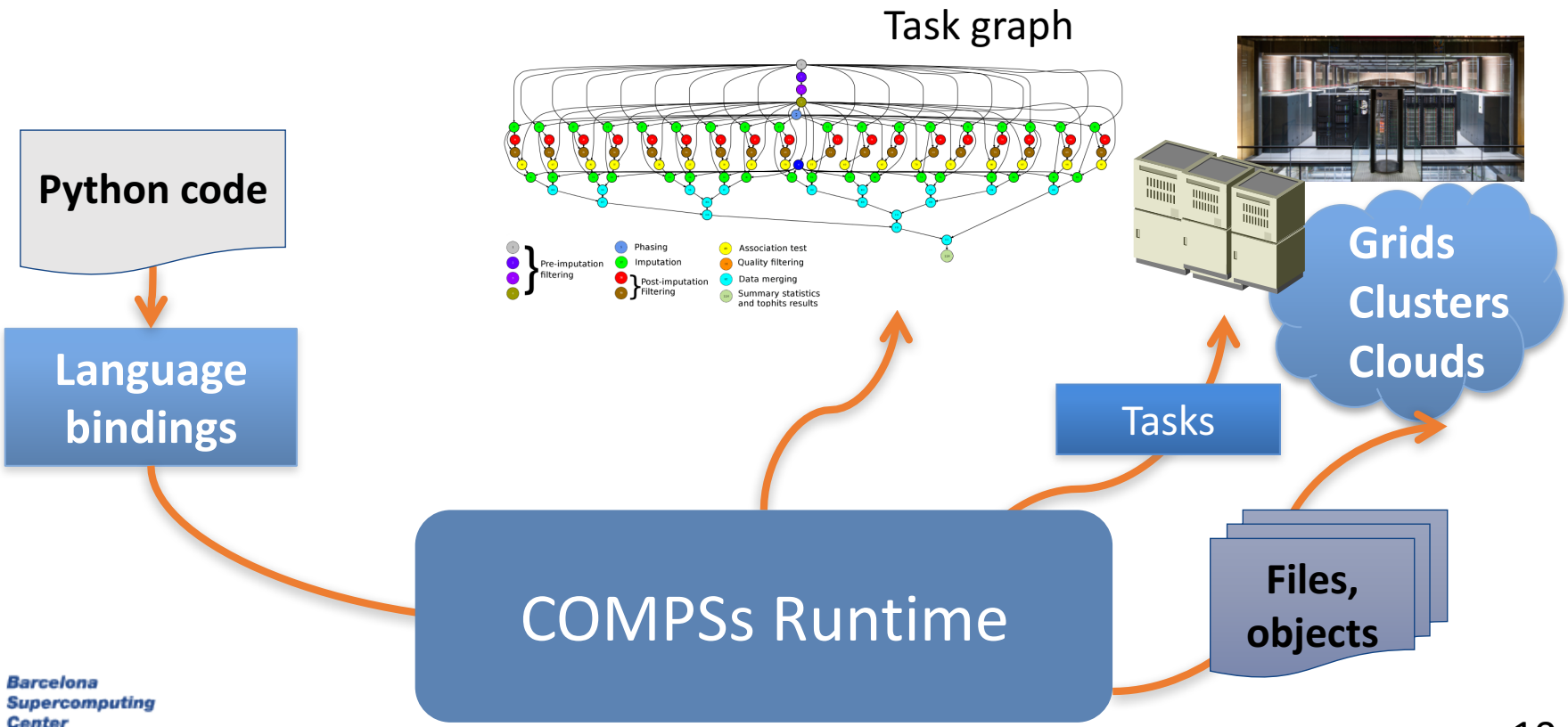
- Contraints enable to define the resource characteristics where to execute the task
- Versions. Mechanism to support multiple implementations of a single behavior
  - Combined with constraints to select different architectures
- Runtime selects more adequate version – depends on actual scheduling policy

```
@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
```

```
@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
```

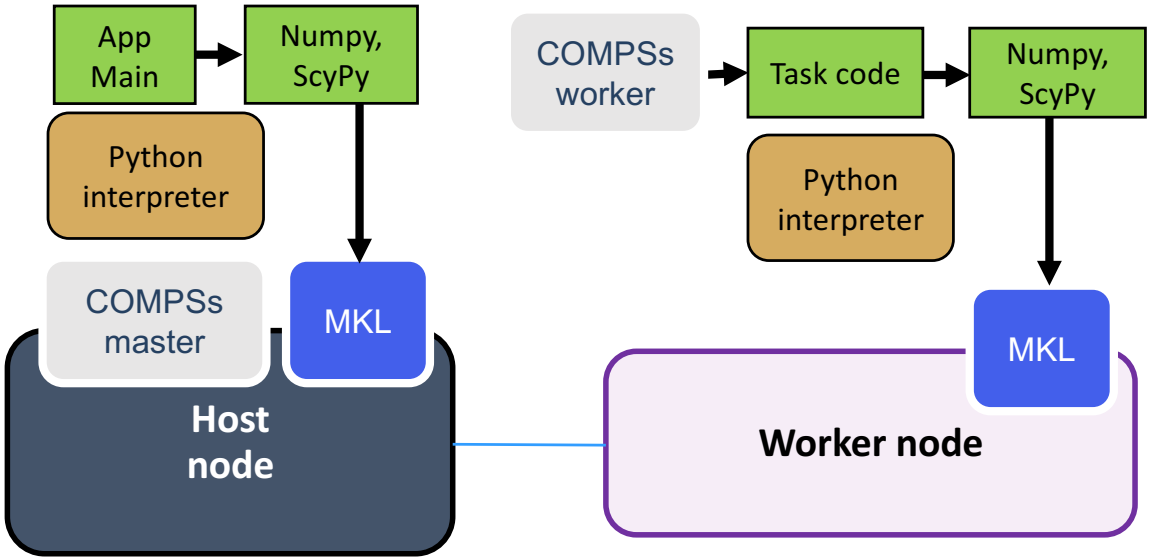
PyCOMPSs runtime

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



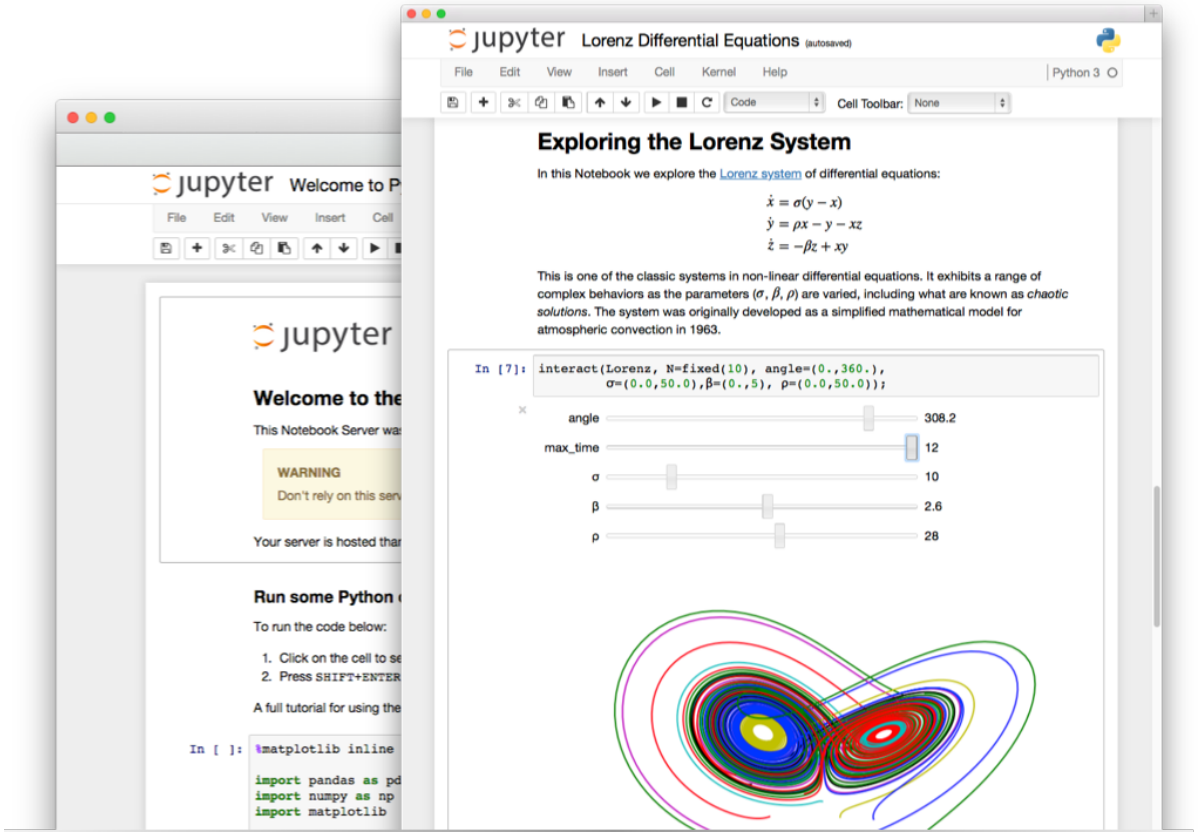
PyCOMPSs stack + Numpy and MKL

- 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



Integration 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



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

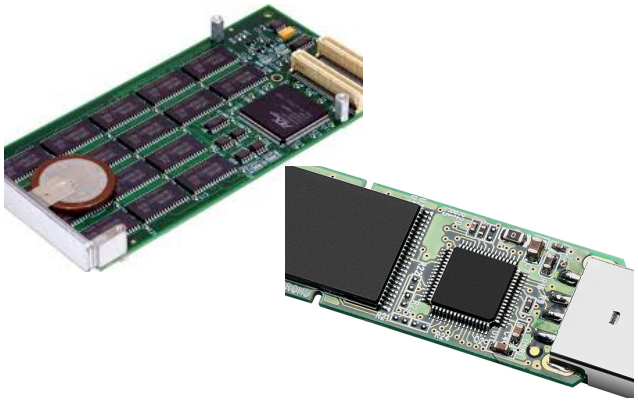
www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

Integration with new storage technologies

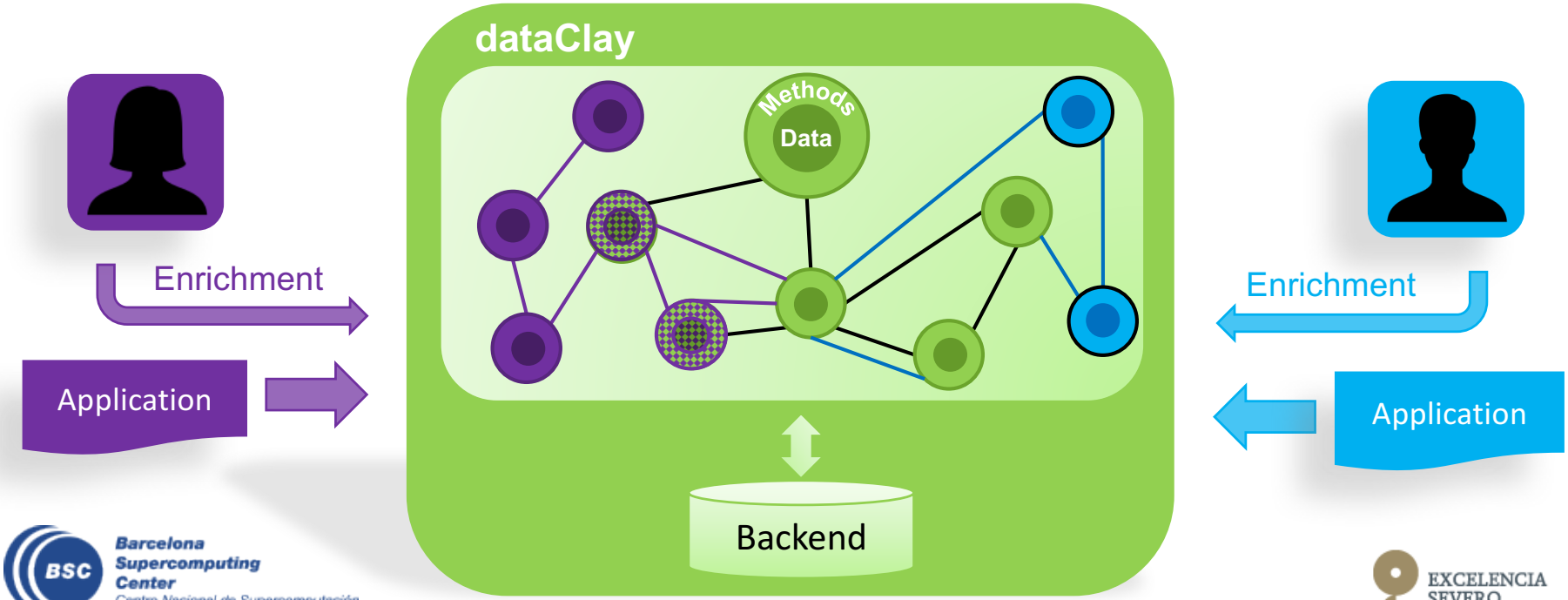
Revolution in the storage technologies



- New storage hardware
  - Non-Volatile RAM
  - Storage Class Memories
- Resemble more memory than storage
  - Low latency, high bandwidth
  - Byte-addressable interface
  - Using them as block devices for a file system does not seem to be the best option
- New storage methodologies are required
  - May imply a disruption on how data is accessed
  - There is no reason to access persistent data in a different way of how non-persistent data is
- New requirements from applications with regard (Big) Data management
  - Large amounts of data generated by simulations and applications in general
  - Moving data is expensive (time, power): in-situ processing

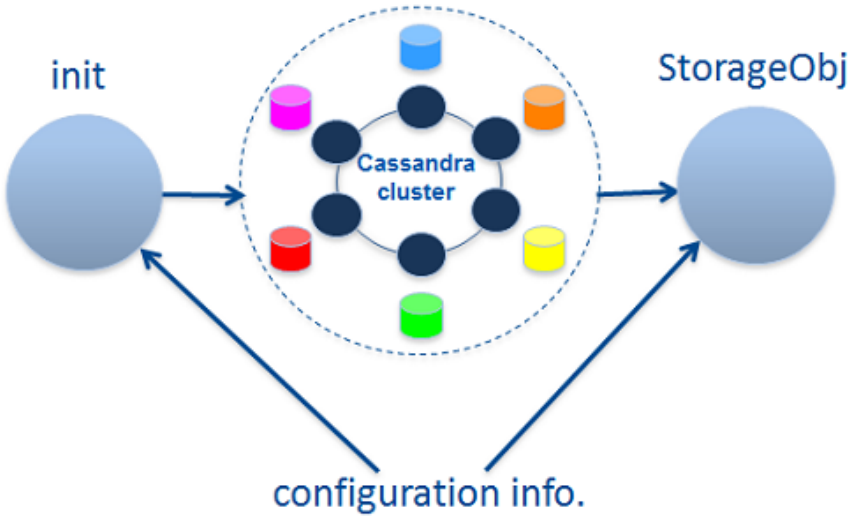
dataClay

- 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



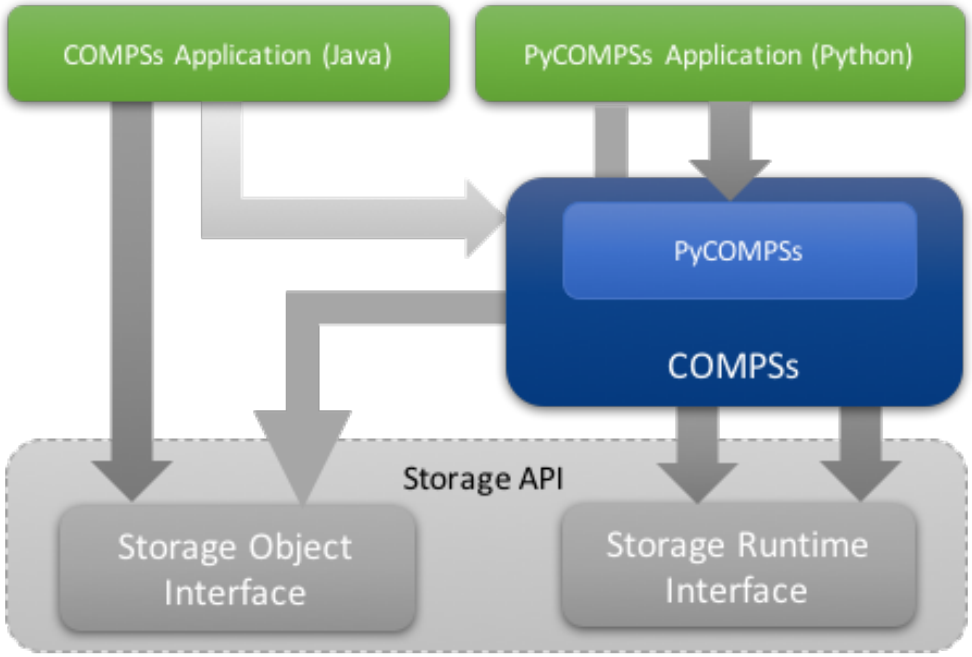
Hecuba

- Set of tools and interfaces that aim to facilitate an efficient and easy interaction with non-relational data-bases
- Currently implemented on Apache Cassandra database
  - However, easy to port to any non-relational key-value data store
- Mapping of Python dictionaries into Cassandra tables
  - Both consist on values indexed by keys
  - Only Python data type supported right now
- Redefinition of Python iterators
  - Accessing blocks of keys
  - Exploiting locality



Integration with Storage: Storage API

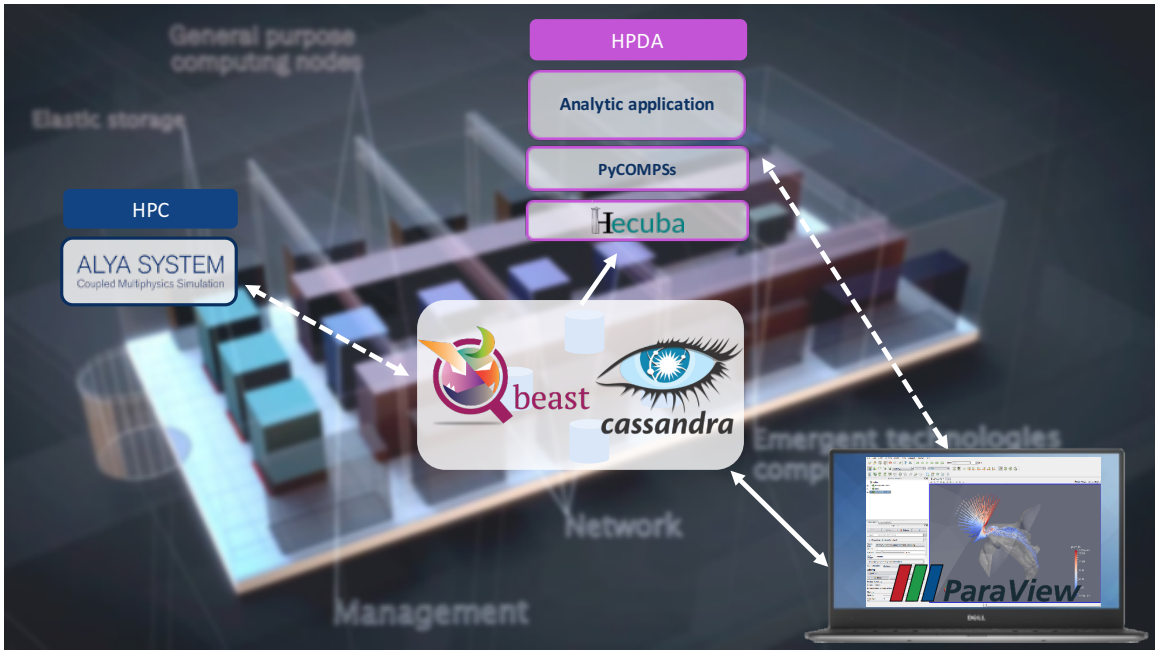
- Integration of programming model with new storage management platforms
- Data made persistent, application agnostic of this persistency
- Producer-consumer
- In-situ




```
...
for i in range(n_points // self.points_per_
    np.random.seed(base_seed + i)
    fragment = Fragment(dim=self.dim,
                        points=np.random.random([self.points_per_fragment, self.dim]),
                        base_index=i * self.points_per_fragment)
    fragment.make_persistent(dest_stloc_id=storage_locations.next())
    self.fragments.append(fragment)
```

Case of study with Hecuba: Respiratory system simulator

- Alya simulation of air flows through the respiratory system.
- Prototype demo implemented on top on key-value data store:
  - Particles generated by simulation stored in Cassandra
  - Managed by Hecuba
- Qbeast: D8-tree index distributed engine
  - Data access with linear scalability
- Queries parallelized with PyCOMPSs
- Visualization and queries simultaneous to simulation
- Demos 7



ISC High Performance
The HPC Event.

SC17

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

Applying to linear algebra examples

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
```

main

```
startTime = time.time()
initialize_variables()
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)
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

```
@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
```

Initialization

```
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)
```

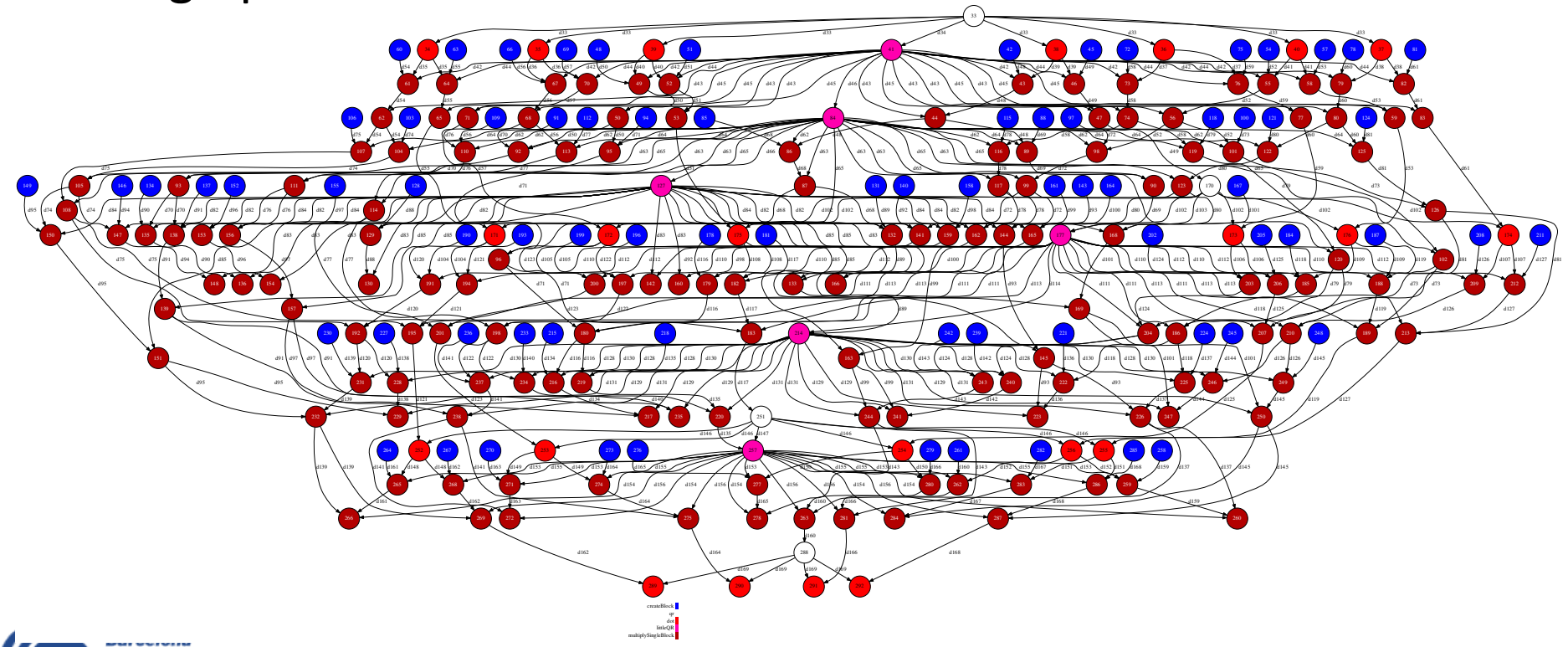
QR factorization

- QR based on Givens rotations, which access the data in the matrices by blocks
- Makes use of an auxiliary matrix
- Blocks are NumPy arrays and operate on NumPy operations

```
def qr_blocked(R):
    Q = genIdentity(MSIZE,BSIZE)
    for i in range(MSIZE):
        actQ, R[i][i] = qr(R[i][i], transpose=True)
        for j in range(MSIZE):
            Q[j][i] = dot(Q[j][i], actQ, transposeB=True)
        for j in range(i+1,MSIZE):
            R[i][j] = dot(actQ,R[i][j])
        #Update values of the respective column
        for j in range(i+1,MSIZE):
            subQ = [[np.matrix(np.array([0])),np.matrix(np.array([0]))],
                    [np.matrix(np.array([0])),np.matrix(np.array([0]))]]
            subQ[0][0],subQ[0][1],subQ[1][0],subQ[1][1],R[i][i],R[j][i] =
                    littleQR(R[i][i],R[j][i], BSIZE, transpose=True)
            #Update values of the row for the value updated in the column
            for k in range(i + 1,MSIZE):
                [[R[i][k]],[R[j][k]]] = multiplyBlocked(subQ, [[R[i][k]],[R[j][k]]],
                    BSIZE)
            for k in range(MSIZE):
                [[Q[k][i], Q[k][j]]] = multiplyBlocked([[Q[k][i], Q[k][j]]], subQ,
                    BSIZE, transposeB=True)
    return Q,R
```

QR factorization

- QR based on Givens rotations, which access the data in the matrices by blocks
- Makes use of an auxiliary matrix
- Task graph for a 4 x 4 blocks matrix



QR factorization: Memory save

- Auxiliary matrix: only those blocks different to identity or zero are allocated
- Main code remains the same

```
@constraint (ComputingUnits="${ComputingUnits}")
@task (returns=list)
def createBlockTask(BSIZE):
    setMKLNumThreads(mkl_threads)
    return np.matrix(np.random.random((BSIZE, BSIZE)), dtype=np.double,copy=False)
```

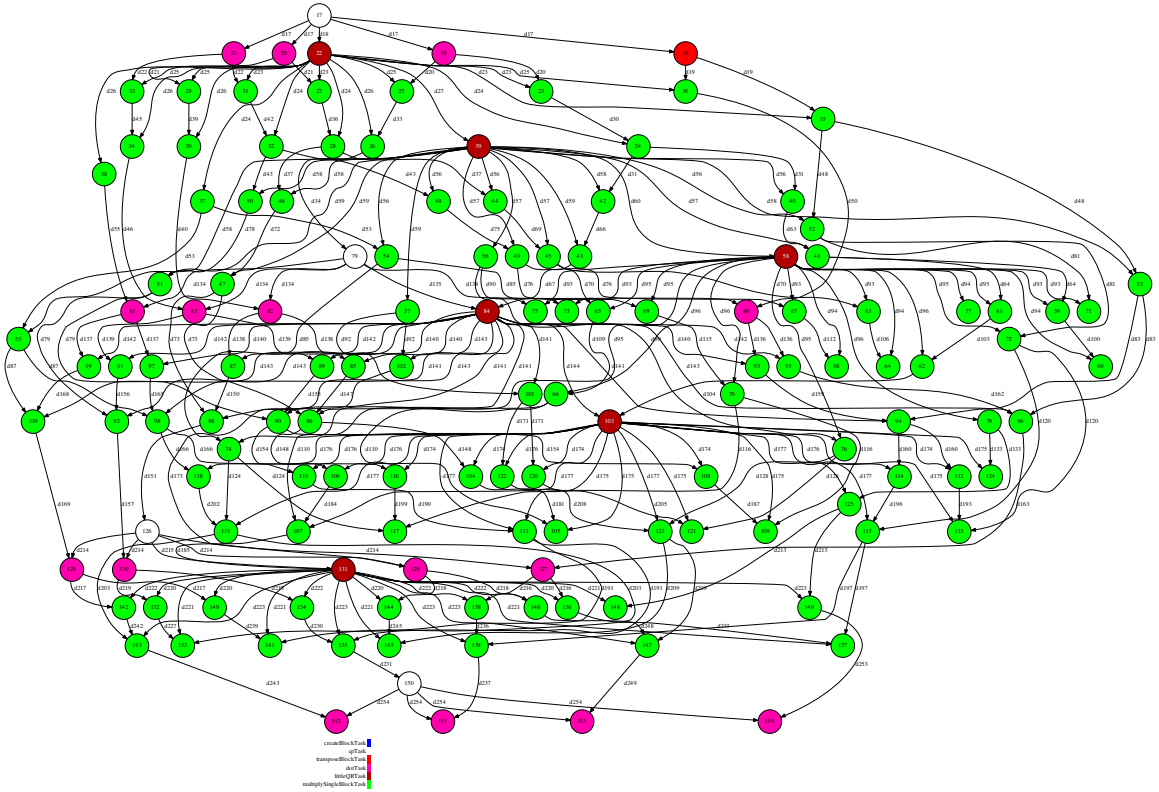
```
def createBlock(BSIZE, type='random'):
    if type == 'zeros':
        block = []
    elif type == 'identity':
        block = []
    else:
        block = createBlockTask(BSIZE)
    return [type,block]
```

```
@constraint (ComputingUnits="${ComputingUnits}")
@task (returns=list)
def transposeBlockTask(A):
    return np.transpose(A
```

```
def transposeBlock(A):
    if A[0] == 'zeros' or A[0] == 'identity':
        return A
    return ['random', transposeBlockTask(A[1])]
```

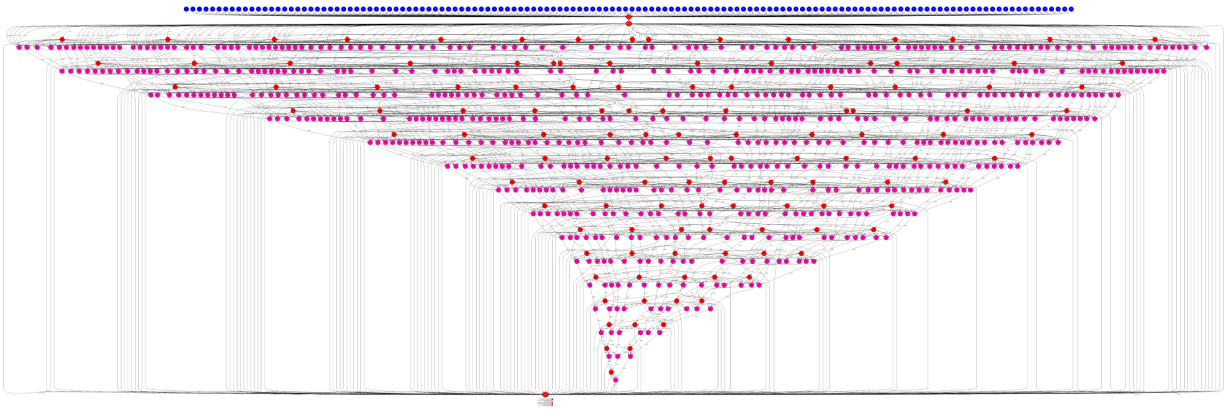
QR factorization: Memory save

- Auxiliary matrix: only those blocks different to identity or zero are allocated
- Main code remains the same
- Task graph for a 4 x 4 blocks matrix

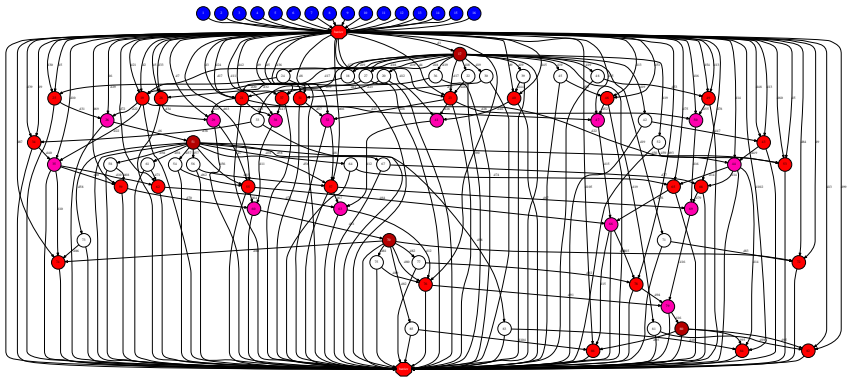


Other examples

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



- LU - Initial implementation with partial pivoting at block level
  - Task graph for a 4 x 4 blocks factorization



www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

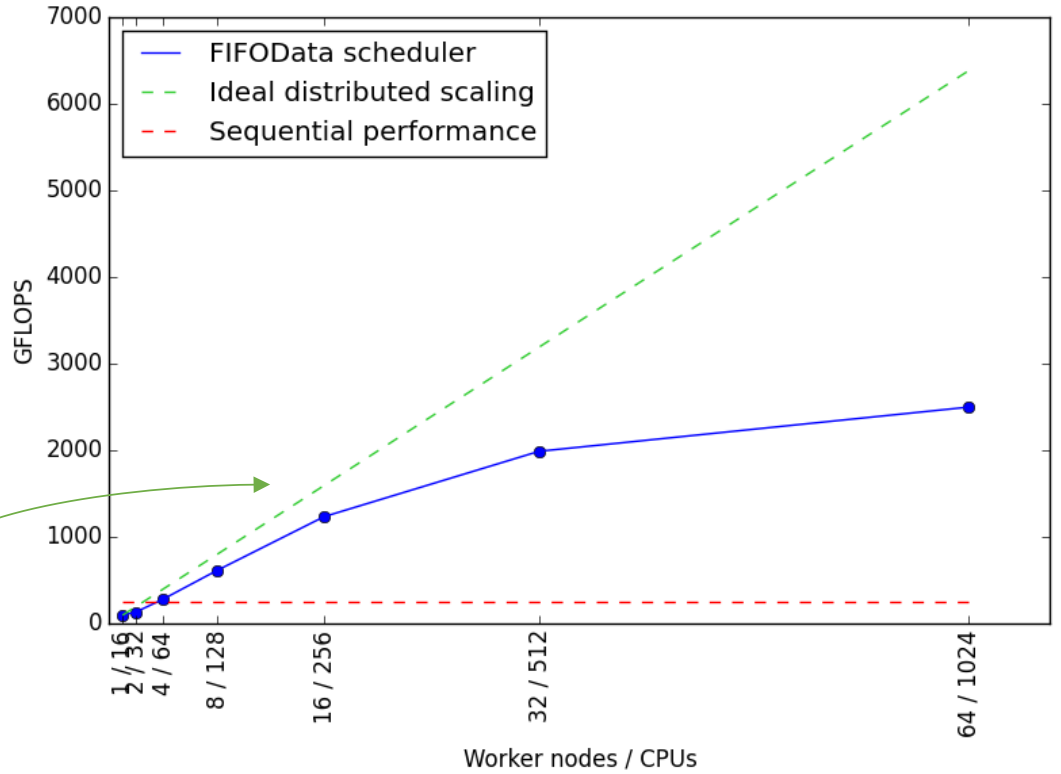
Performance results in several architectures

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 MKL threads / task
oversubscription = 4

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



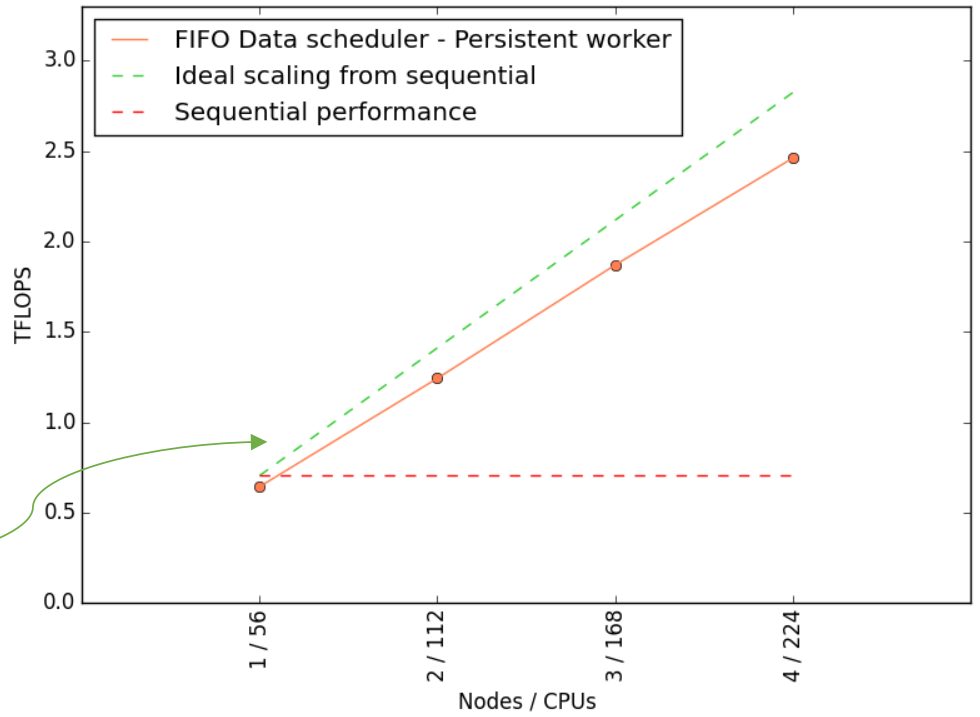
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Matmul evaluation

- Results in SSF cluster, distributed through several nodes
  - Lustre as shared file system
  - 2 x Xeon 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 MKL threads / task
No oversubscription
Master in first worker node

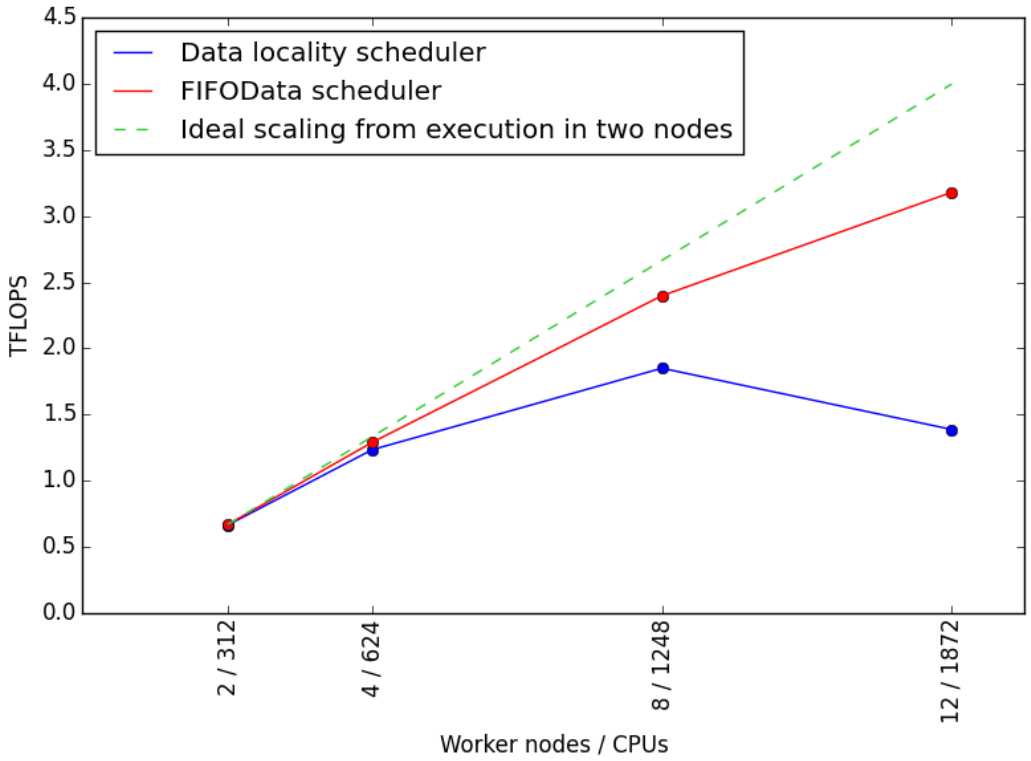
Baseline: Maximum performance
Obtained with multi-threaded
MKL in a single node
with a block of size 16384 x 16384



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Matmul evaluation - heterogeneous nodes

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



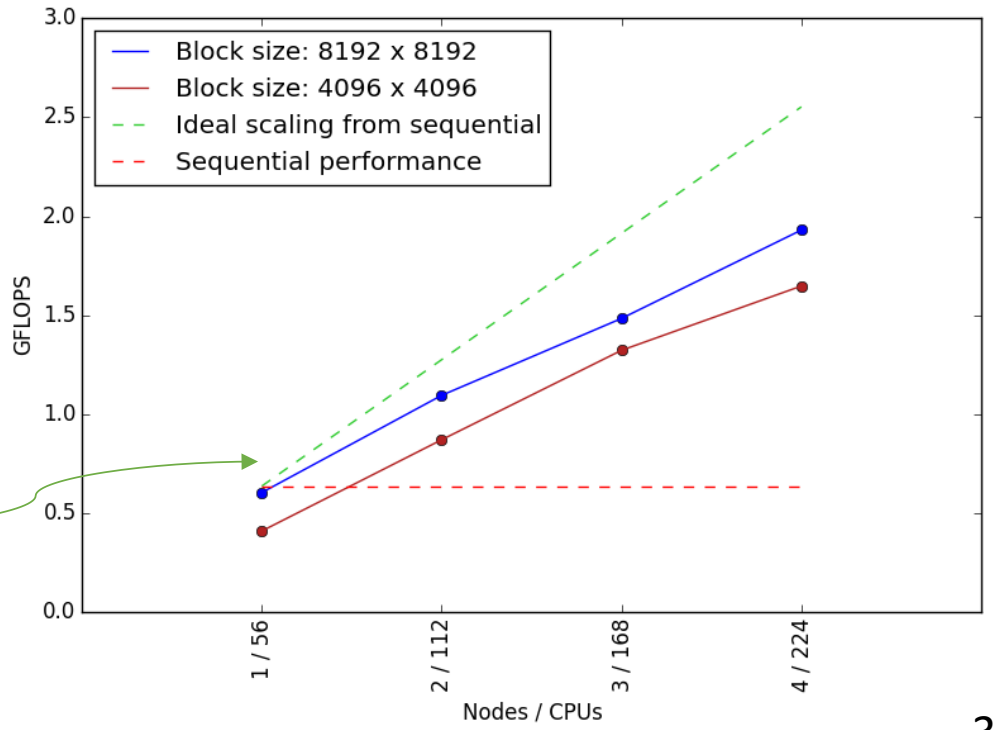
Matrix = 131 K X 131K doubles
Xeons: 8 tasks per node, 7 threads per task
KNL: 4 tasks per node, 64 threads per task
IntelPython 2.7.13
Block size: 4096 x 4096

Cholesky evaluation

- Results in SSF cluster, distributed through several nodes
  - 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)
IntelPython 2.7.13

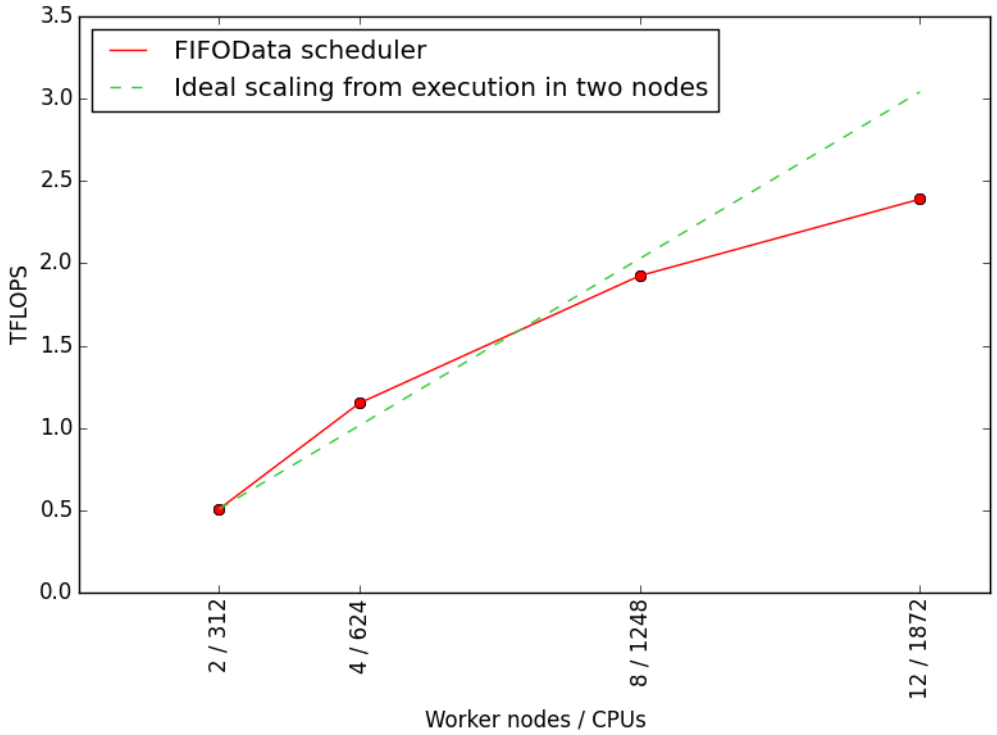
Baseline: Maximum performance obtained with multi-threaded MKL in a single node for a block of size 8192 x 8192



Cholesky evaluation– heterogeneous nodes

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

Matrix = 128K X 128K doubles
Xeons: 8 tasks per node, 7 threads per task
KNL: 4 tasks per node, 64 threads per task
IntelPython 2.7.13
Block size: 4096 x 4096



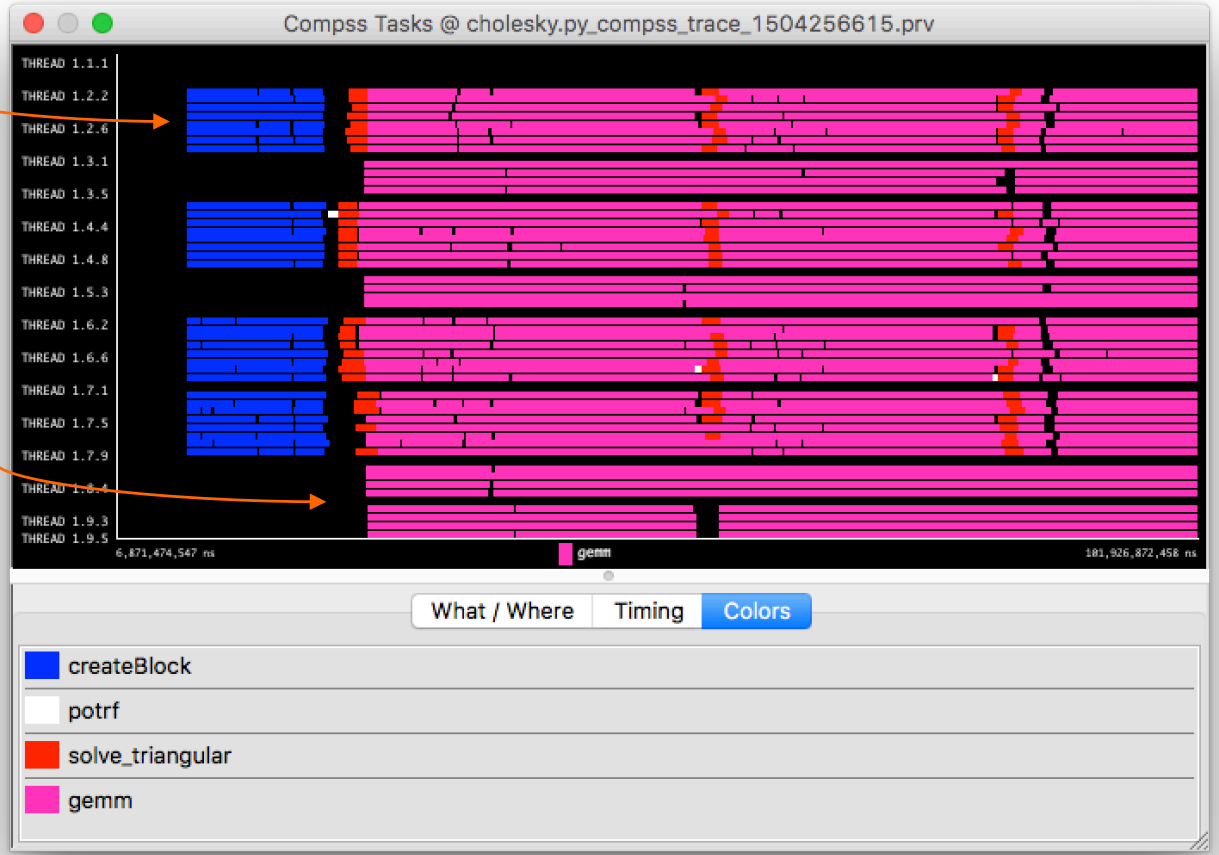
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Cholesky evaluation– heterogeneous nodes

- Timeline of tasks
- KNL nodes only execute dgemms
  - Selected with constraints decorator

Xeon node

KNL node

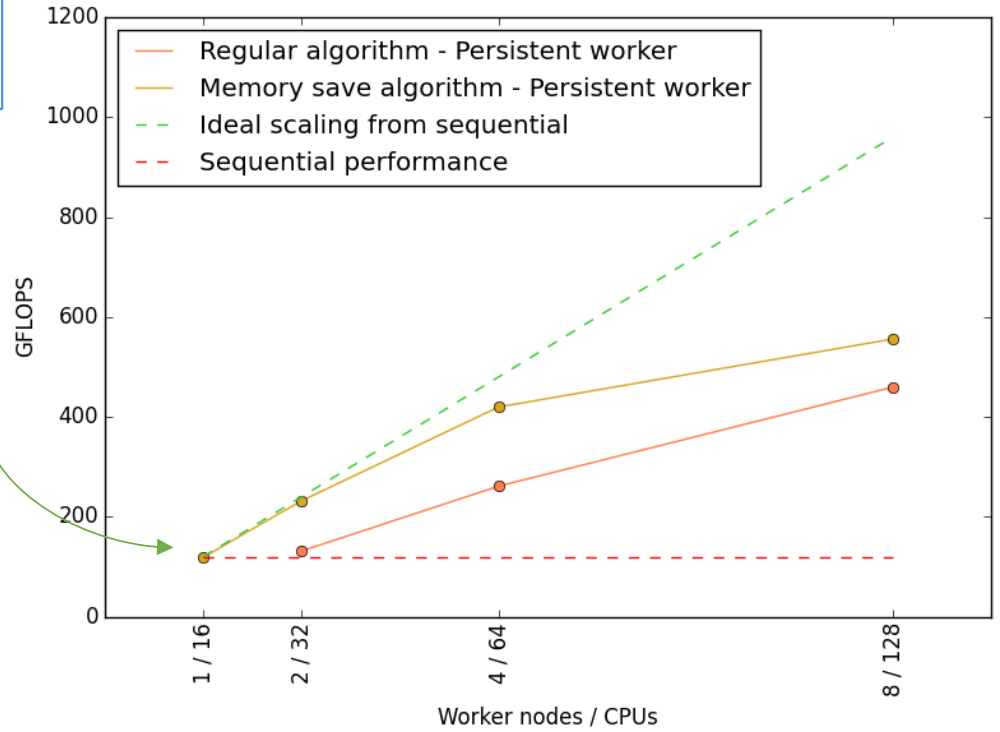


QR results: comparing algorithms

- Results in MareNostrum III, distributed through several nodes
  - Intel SandyBridge-EP E5-2670

Matrix = 32K X 32K doubles
Block size = 4096 x 4096 doubles
4 tasks per node
16 MKL threads per task (oversubscribing = 4)
IntelPython 2.7.13

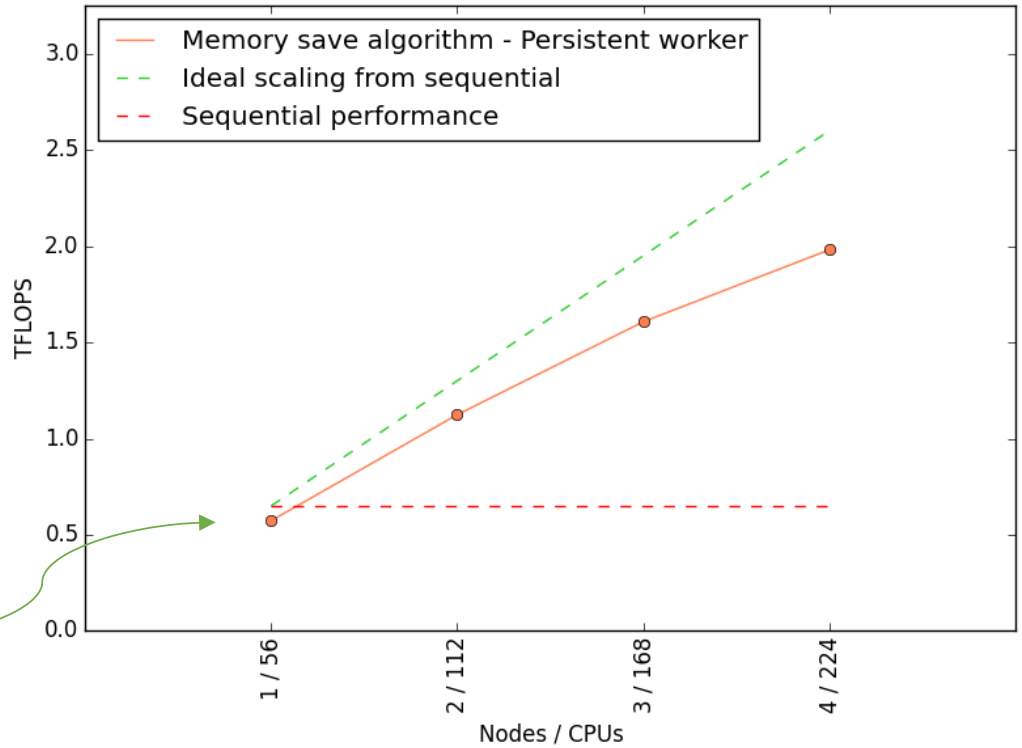
Baseline: one MKL QR
one block with 16 threads



QR memory-save results

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

Matrix = 56K X 56K doubles
Block size = 4096 x 4096 doubles
8 tasks per node
7 MKL threads per task (oversubscribing = 1)
IntelPython 2.7.13



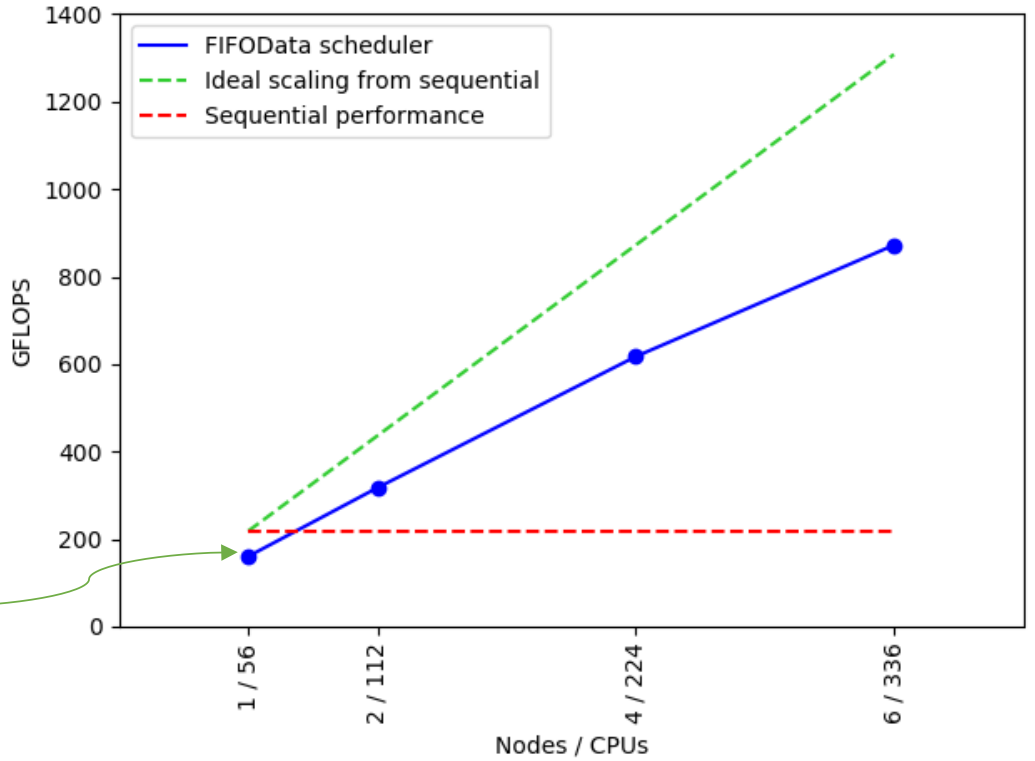
Baseline: Maximum performance attained with multi-threaded MKL in a single node
a block of size 32768 x 32768

LU evaluation

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

Matrix = 82K X 82K doubles
Block size = 4096 x 4096 doubles
8 tasks per node
7 MKL threads per task (oversubscribing = 1)
IntelPython 2.7.13

Baseline: Maximum performance attained with multi-threaded MKL in a single node
a block of size 32K x 32K



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

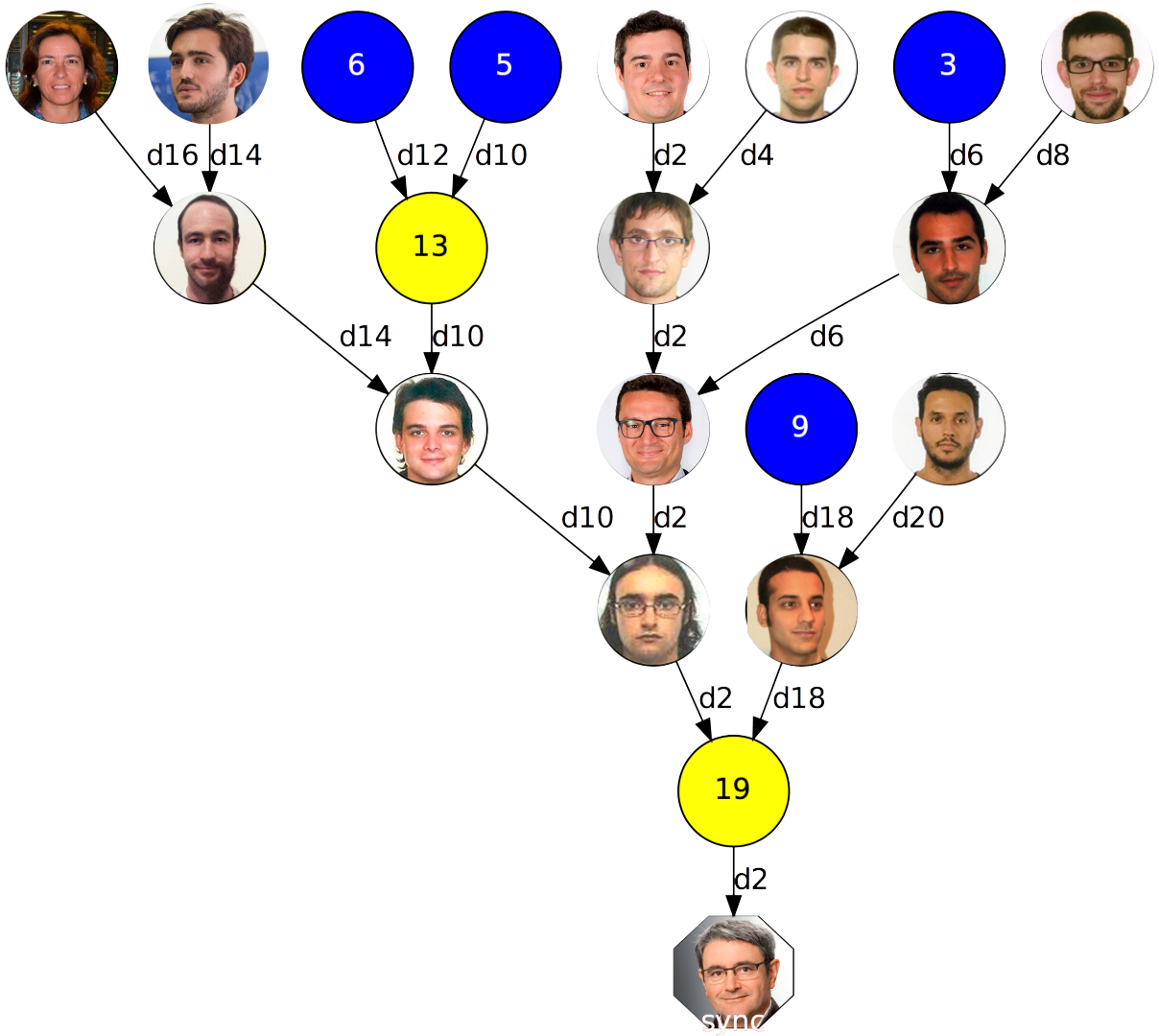
Conclusions and future work

Conclusions and future work

- Task-based programming models provide a friendly interface that enables the parallelization of sequential applications
- PyCOMPSs provides a high level, easy interface to develop parallel linear algebra codes
  - Sequential Python scripts
  - Two level of parallelism: Parallelisation at task level and thread parallelization (MKL)
  - Agnostic of the platform
  - Possibility to run Python sequential codes in distributed platforms with small effort
    - Performance reasonable
    - Use of distributed memory
- Tradeoff between scalability and performance
  - Performance analysis with Paraver helps significantly
    - PyCOMPSs traces with hardware counters
- Future work:
  - Enhacenments in scheduling policies
  - Develop reduction schemes
  - New codes and try new architectures
  - Implement data analytics algorithms on top of PyCOMPSs, using linear algebra parallelized kernels as basis

Workflows and distributed computing group (COMPSs)

- Adrià Aguilar
- Javier Alvarez
- Pol Alvarez
- Ramon Amela
- Rosa M Badia
- Javi Conejero
- Jorge Ejarque
- Daniele Lezzi
- Francesc Lordan
- Cristian Ramon-Cortés
- Sergio Rodriguez
- Carlos Sagarra
- Albert Serven



www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

Thanks!

For further information please contact

rosa.m.badia@bsc.es

compss.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Intel-BSC Exascale Lab

Collaboration between Intel and the Barcelona Supercomputing Center