Enabling Python to execute efficiently in heterogeneous distributed infrastructures with PyCOMPSs

Ramon Amela
Barcelona Supercomputing Center
Barcelona, Spain
ramon.amela@bsc.es

Cristian Ramon-Cortes
Barcelona Supercomputing Center
Barcelona, Spain
cristian.ramoncortes@bsc.es

Jorge Ejarque
Barcelona Supercomputing Center
Barcelona, Spain
jorge.ejarque@bsc.es

Javier Conejero
Barcelona Supercomputing Center
Barcelona, Spain
francisco.conejero@bsc.es

Rosa M. Badia
Barcelona Supercomputing Center
Barcelona, Spain
Consejo Superior de Investigaciones Cientificas (CSIC)
Barcelona, Spain
rosa.m.badia@bsc.es

ABSTRACT

Python has been adopted as programming language by a large number of scientific communities. Additionally to the easy programming interface, the large number of libraries and modules that have been made available by a large number of contributors, have taken this language to the top of the list of the most popular programming languages in scientific applications. However, one main drawback of Python is the lack of support for concurrency or parallelism. PyCOMPSs is a proved approach to support task-based parallelism in Python that enables applications to be executed in parallel in distributed computing platforms.

This paper presents PyCOMPSs and how it has been tailored to execute tasks in heterogeneous and multi-threaded environments. We present an approach to combine the task-level parallelism provided by PyCOMPSs with the thread-level parallelism provided by MKL. Performance and behavioral results in distributed computing heterogeneous clusters show the benefits and capabilities of PyCOMPSs in both HPC and Big Data infrastructures.

CCS CONCEPTS

• **Software and its engineering** → **Distributed programming languages**; • **Mathematics of computing** → *Computations on matrices*;

KEYWORDS

HPC, Python, Big Data, Linear Algebra, Heterogeneous infrastructures

ACM Reference Format:
Ramon Amela, Cristian Ramon-Cortes, Jorge Ejarque, Javier Conejero, and Rosa M. Badia. 2018. Enabling Python to execute efficiently in heterogeneous distributed infrastructures with PyCOMPSs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

1 MOTIVATION

Python is a duck typed, object oriented, interpreted and easy to learn programming language that plays well with other languages (i.e. with C) [35]. Python has been adopted by a large community, including scientific communities from lots of different areas. This adoption is not only due to the language features but also because of the large amount of third-party libraries available for the community, such as NumPy [36] and SciPy [26], that offer vectorized data structures and numerical routines. NumPy automatically maps operations on vectors and matrices to the BLAS [9] and LAPACK [13] functions present in the system. Indeed, NumPy is a *de-facto* standard when working with tensors in Python due to the high performance achieved and its ease of use. When a multi-threaded BLAS version is present in the system (using OpenMP [19] or TBB [10]), the operations are automatically parallelized.

As mentioned above, Python is an interpreted language, being CPython its most common interpreter. A very well-known limitation of CPython is the use of a Global Interpreter Lock (GIL) which disables concurrent Python threads within one process. This basically means that, although threads are supported in Python, only one will execute at a time.

Alternatives to provide parallelism in Python are the multiprocessing, Parallel Python (PP) and MPI modules. The multiprocessing module provides [2] support for the spawning of processes in SMP machines using an API similar to the threading module, with explicit calls to create processes. On the other hand, the Parallel Python (PP) module [3] provides mechanisms for parallel execution of Python codes, with an API with specific functions to specify the number of workers to be used, submit the jobs for execution, get the results from the workers, etc. Finally, the mpi4py [20] library allows the user to open parallelism both inter-node and intra-node. In all cases, the management of the parallelism is the programmer's responsibility.

PyCOMPSs [34] is a task-based programming model that offers an interface based on Python sequential paradigm. It enables the execution in parallel by means of building, at execution time, a data dependency graph of the tasks that compose the application. The syntax of PyCOMPSs is minimal, using decorators to enable

the programmer to identify those methods that are application tasks and a small API for synchronization. PyCOMPSs relies on a runtime that can exploit the inherent parallelism at task level and execute the application in a distributed parallel platform (clusters and clouds). The runtime is responsible for scheduling the tasks in the available computation resources, performing the necessary data transfers between distributed memory resources when no shared data system is available, synchronizing all activities, acting as an interface with different computing resources such as cloud middlewares, etc.

There are similar libraries and frameworks that enable Python distributed and multi-threaded executions such as Dask [21] and PySpark [11]. Dask is a native Python library that allows both the creation of custom DAG's and the distributed execution of a set of operations on NumPy and pandas [30] objects. PySpark is a binding to the widely extended framework Spark [37]. A previous paper compares several Big Data algorithms using the native version of both COMPSs and Spark runtimes [18].

This paper presents PyCOMPSs functionalities through numerical codes such as matrix multiplication and several matrix factorizations. The main contributions are the extensions to the PyCOMPSs programming model to support multi-threaded libraries, a new scheduling infrastructure, and the support for multiple tasks' versions. This set of extensions allows to achieve good performance with a moderate encoding effort, enabling not expert users to reach HPC behaviors even under Big Data conditions (i.e. when data is already present in the infrastructure instead of being initialized for the computation). Moreover, the same code can be executed in a multi-threaded way on a single machine, in the cloud or an heterogeneous cluster, making it highly portable.

Another contribution of this paper is a set of linear algebra kernels written in PyCOMPSs, which conform a prototype of a parallel linear algebra library that would be made available for the community in the near future.

The rest of the paper is organized as follows: Section 2 presents PyCOMPSs and its features, Section 3 describes the linear algebra kernels, Section 4 presents performance results, and Section 5 concludes the paper and gives some guidelines for future work.

2 PYCOMPSS OVERVIEW

PyCOMPSs is a task-based programming model that aims to make easier the development of parallel applications, targeting distributed computing platforms. *Tasks* are identified by the programmer using simple annotations in the form of Python decorators, which indicate that invocations of a given method will become tasks at execution time. The `@task` decorator also contains information about the directionality of the method parameters specifying if a given parameter is read (IN), written (OUT) or both read and written in the method (INOUT).

Figure 1 shows an example of task annotation. In the example, the parameter `c` is of type INOUT. The directionality tags are used at execution time to derive the data dependencies between tasks and are applied at object level, taking into account its references to identify when two tasks access the same object. The priority tag is a hint for the PyCOMPSs' scheduler that will then execute the tasks with this tag earlier, always respecting the data dependencies.

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

Figure 1: Sample task annotation

Additionally to the `@task` decorator, the `@constraint` decorator can be optionally defined to indicate some task hardware or software requirements. In Figure 1, the task constraint `ComputingUnits` shows to the runtime how many resources are consumed by each task execution. The available resources are defined by the system administrator in a separated `XML` configuration file. Other constraints that can be defined refer to processor architecture, memory size, etc. A tiny synchronization API completes the PyCOMPSs syntax. Concerning the example presented in Figure 2, the API function `compss_wait_on` waits until all the tasks modifying the `result`'s value are finished and brings the value to the master's memory. Once the value is retrieved, the execution of the master's code is resumed. Given that PyCOMPSs is used mostly in distributed environments, synchronizing may imply a data transfer from a remote storage or memory space to the master node.

```
for block in Data:
    presult = word_count(block)
    reduce_count(result, presult)
finalResult = compss_wait_on(result)
```

Figure 2: Sample call to synchronization API

Since the PyCOMPSs runtime is written in Java[28], Python syntax is supported using a binding. This Python binding is supported by the Binding-commons layer which focuses on enabling the functionalities of the runtime to other languages (currently, Python and C/C++). It has been designed as an API with a set of defined functions. It is written in C and performs the communication with the runtime through the JNI [27]. The main advantage of this architecture is the capability to execute the code across several heterogeneous architectures. The user defines the tasks but is the runtime who handles all the infrastructure thanks to the several available connectors.

2.1 PyCOMPSs runtime

The PyCOMPSs runtime handles the execution of the applications in the computing infrastructure. The computing infrastructure is composed of several heterogeneous nodes, and the execution is orchestrated following a master-worker paradigm, where the main program is started on the master node and tasks are offloaded to worker nodes. In the most general case, the node allocating the master node will also allocate a worker node.

Once the application starts, each time a task is invoked, a node is added to the task graph. The directionality of the parameters is used to identify the data dependencies between the new task and previous ones. Each scheduler will analyze the generated graph in a particular way to execute all the workload among the available resources. The runtime is also aware of the location of the different data objects and files in the distributed computing platform. This

information is taken into account when scheduling to exploit the *data locality* and also to decide which objects must be transferred between different memory spaces to guarantee that all the tasks can be executed.

When transferences between different memory spaces are needed, the PyCOMPSs binding serializes and writes the objects to disk using the standard library `Pickle`, delegating the transfer to the runtime.

Furthermore, the available computing units seen by the runtime can be configured to oversubscribe the amount of work, meaning that more threads than CPUs are created.

2.2 Interaction with external libraries

The PyCOMPSs runtime supports the execution of multi-threaded tasks using the constraint interface. The number of cores assigned to a multi-threaded task can be indicated by the programmer in the `ComputingUnits` constraint tag. The PyCOMPSs scheduler can assign several cores to a given multi-threaded task. On the other hand, although support for tasks that use several nodes has been added recently, in this work we only consider tasks executing inside a single node.

Before this work, the cores were assigned blindly to the tasks. The performance results observed were relatively poor when running numerical applications, such as those using the NumPy or SciPy libraries that link to the Intel®MKL library [1]. It has been shown that, by default, MKL tends to occupy the entire node when the multi-threading is enabled. Not considering this fact can result in a heavy oversubscribing. In addition, each task can be executed in several NUMA sockets. This fact increases the amount of transfers between the different NUMA-nodes, decreasing the performance dramatically. Knowing that this behavior can be found in other libraries, the problem has been solved in a general way.

We have modified the PyCOMPSs task executor in such a way that it is currently able to bind multi-threaded tasks to specific computing units of the infrastructure. This fact, combined with the capability to define the nodes' virtual amount of computing units, allows the user to achieve the desired rate of oversubscribing. However, this is not done blindly: the PyCOMPSs runtime distributes the tasks evenly between the different NUMA sockets, avoiding at the maximum the transfers between memory spaces.

2.3 Scheduling infrastructure

PyCOMPSs runtime has been extended with a scheduling infrastructure that supports pluggable scheduling policies. Almost all the tests presented in this paper are based on a *data locality* scheduler that takes into account the node that stores the data accessed by the tasks. More precisely, a task will have a score equal to the amount of input data present in a given node.

Defining a new score policy is enough to change the scheduler behavior. It will prioritize the tasks with the highest score for a given combination of resource, implementation, and data. In addition to the *data locality* score, three more policies have been defined: First In First Out (FIFO), Last In First Out (LIFO) and *data locality* with priority to tasks with a shared edge in the dependency graph with the finished task (FIFOData). In this last policy, there are two different scenarios. In the case where there are tasks freed by the job that has just finished, one of them is scheduled in First In First Out order; even before treating the tasks that are already free. Otherwise, *data locality* is considered between all the available tasks. The first two policies (LIFO, FIFO) have served to probe the robustness of the scheduling system. The third one can be seen as a relaxation of the *data locality* scheduler to lighten the amount of needed comparisons to schedule a task.

The available schedulers allow the user to configure the execution depending on the expected load.

2.4 Python persistent workers

In previous runtime versions, PyCOMPSs was enhanced with a persistent Java worker, meaning that a Java worker process was started at the beginning of the application execution, communicating with the master to get information about the tasks to be executed and data transfers to be performed. However, every time a Python task was invoked, a new Python interpreter was launched. This process has been enhanced with the implementation of Python persistent workers.

More in detail, the PyCOMPSs worker module has been modified on top of the Python's built-in multiprocessing library. When the application execution begins, the primary worker process in each worker node spawns a set of processes that will be responsible for executing the tasks. These processes are kept alive during the whole application execution and communicate with the Java persistent worker through pipes. The messages that they exchange include information about the task execution requests, job parameters, and computation results. This feature improves the overall performance by reducing the overhead of deploying a new Python interpreter per task. Besides, modules loaded by previous tasks are already present in the interpreter and do not need to be reloaded.

2.5 Versioning

GPUs have demonstrated that can sometimes achieve better performance than CPUs. In fact, it is not always easy to decide whether it is better to use one architecture or the other[17]. Also, FPGAs are gaining some momentum. In this context, projects with the primary focus of interest on heterogeneous architectures are arising[23]. Hence, it seems reasonable to think that, in both HPC and Big Data contexts, we are going towards environments with heterogeneous architectures.

```
@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
```

Figure 3: Version handling with PyCOMPSs

PyCOMPSs can manage those cases by providing support for the definition of different versions of the same method for different architectures. The programmer can use the `@implements` decorator to indicate that a method implements the same behavior than another. Figure 3 shows an example of versioning, which together with the `@constraint` decorator allows to indicate to the runtime that some tasks can only be executed in a given set of computing resources. In fact, using versioning and tasks' constraints, the users can define CPU, GPU or FPGA versions of the same task.

Internally, at the beginning of the execution, the Runtime will blindly execute any of the available versions that can run in a given resource in order to obtain an execution profile per version. Afterwards, the Runtime is capable to use the profiled information to choose the implementation with the lowest execution time.

2.6 Profiling

PyCOMPSs generates *post-mortem* traces under demand using Extrae[4]. These files can be explored with Paraver[31] [5], obtaining visual information to make easier the code performance fine tuning.

Some specific PyCOMPSs events have been added in order to differentiate the different steps done by the master and the workers. More precisely, it is possible to see the different actions performed by a worker each time that a task is executed.

Finally, the dependency graph generated can be plotted at the end of the computation or be explored on runtime with the monitor.

3 LINEAR ALGEBRA CODES

We have evaluated PyCOMPSs with several linear algebra codes. It is important to keep in mind that PyCOMPSs is a general purpose programming model, not a specific one for dense linear algebra computations [16] [12]. Nevertheless, linear algebra algorithms are the base for several fields such as Machine Learning and Computational physics. Even if other good options like ScaLAPACK [15] already do this job, any of them can be called directly from Python.

In general, the matrices are chunked in smaller square matrices (*blocks*) to distribute the data easily along the available resources and take advantage of this fact to consider the square blocks as the minimum entity to work with [25]. All the operations performed on the blocks use the multi-threaded library `MKL`.

The following schema has been pursued for all the computations. The initialization is performed in a distributed way, defining tasks to initialize the matrix blocks. These tasks do not take into account the nature of the algorithm and they are scheduled in a round robin manner. Next, all the computations are done considering that the data is already allocated in a given node. The data locality scheduler will assign the tasks taking into account the locality information and reducing the data transfers. This methodology shows that PyCOMPSs is capable of obtaining HPC performance in Big Data environments, where data is already present in the infrastructure, and it is not possible to arrange its location depending on the computation.

The following subsections provide a brief description of the encoded algorithms.

3.1 Matrix multiplication

The matrix multiplication code performs a matrix multiplication by blocks. The code has two tasks: one for the creation of the matrices' blocks and one for the blocks' multiply-accumulate. Since the blocks are defined as NumPy arrays, the operations that operate on them are overridden and the corresponding NumPy operation is invoked, which calls as well to the MKL operation. Notice that this behavior happens even when indicated with arithmetic operators.

Figure 1 shows the code used to perform the multiplication task.

3.2 Cholesky factorization

The Cholesky factorization can be applied to Hermitian positive-defined matrices. This decomposition is a particular case of the LU factorization, obtaining two matrices of the form $U = L^t$.

```
def cholesky_blocked(A):
    import os
    import numpy as np
    for k in range(MSIZE):
        # Diagonal block factorization
        A[k][k] = potrf(A[k][k], mkl_threads)
        # Triangular systems
        for i in range(k+1, MSIZE):
            A[i][k] = solve_triangular(A[k][k], A[i][k], mkl_threads)
            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, mkl_threads)
    return A
```

Figure 4: Cholesky factorization main function

There are two main blocked algorithms to perform a Cholesky decomposition. The right-looking algorithm [14] and the left-looking version [29]. Figure 4 shows the code used in this case, corresponding to the right-looking approach. It has been chosen because it is more aggressive, meaning that in an early stage of the computation there are blocks of the solution that are already computed. Hence, the runtime can continue performing the following computations on the matrices.

The functions in bold in the Cholesky code (`potrf`, `solve_triangular`, and `gemm`) are annotated as tasks. Each of these tasks internally calls to MKL functions, with a given number of threads.

```
@constraint(ComputingUnits="$ComputingUnits")
@task(returns=np.ndarray)
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
```

Figure 5: `potrf` task in the Cholesky code

Figure 5 shows the code of the `potrf` task from Cholesky code. This task has a constraint decorator that indicates the number of `ComputingUnits` (CPU's in this case) required to execute the task that, during the experimentation, matches the amount of MKL threads. Notice that the COMPSs runtime can be configured to

oversubscribe the computing nodes with tasks that involve more threads than the actual available computing cores.

The other tasks defined in this example look very similar to the `potrf` one.

3.3 QR factorization

Traditional QR algorithms use the Householder transformation, but this method requires accessing a whole column of the matrix, while our data structures are based on blocks. The approach followed in our implementation uses a method based on Givens rotations, which access the data in the matrices by blocks [33].

```
def qr_blocked(A, MKLProc, overwrite_a=False):
    import numpy as np
    Q = genIdentity(MSIZE,BSIZE,MKLProc)
    if not overwrite_a:
        R = copyBlocked(A)
    else:
        R = A
    for i in range(MSIZE):
        actQ, R[i][i] = qr(R[i][i], MKLProc, BSIZE, transpose=True)
        for j in range(MSIZE):
            Q[j][i] = dot(Q[j][i], actQ, MKLProc, transposeB=True)
        for j in range(i+1,MSIZE):
            R[i][j] = dot(actQ,R[i][j],MKLProc)
        #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],MKLProc,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, MKLProc)
            for k in range(MSIZE):
                [[Q[k][i],Q[k][j]]] = multiplyBlocked([[Q[k][i],
                    Q[k][j]]], subQ, BSIZE, MKLProc,transposeB=True)
    return Q,R
```

Figure 6: QR factorization main function

Figure 6 shows the QR implementation used in this paper. An auxiliary matrix, mainly composed of identity and zero blocks except for the four blocks corresponding to the positions (i, i), (i, j), (j, i) and (j, j), where (i, j) is the position that is being rotated (changed to zero) performs the rotations. Although in our initial version of the QR algorithm we were allocating this auxiliary matrix, we have developed a second version where only those blocks different to identity or zero are actually allocated, and only the multiplications by values different to identity or zero are performed. With this second approach (called *memorySaveQR* in the evaluation), we save memory space and reduce useless computations.

3.4 LU factorization

In this case, an approach without pivoting [24] has been the starting point. The partial pivoting blocked algorithm [32] was not considered because requires an entire column to be present in a node to compute the partial column *LU* decomposition. Knowing that this approach is unstable in general [22], one modification has been done to increase the stability of the algorithm while keeping the block division and avoiding bringing an entire column into a single node.

Figure 7 shows a schema with all the steps taken in an iteration of the LU factorization:

(1) The current principal block, considered the one with the lowest column and row index (A in the figure) is decomposed using some underlying library
(2) The first U's row (U_{12} in the figure) is computed using the row of the current principal block
(3) The column of the present main block is used to calculate an auxiliary result for the next steps ($P_{22}L_{21}$)
(4) The LU decomposition of the blocks with a row or column index larger than the principal one is launched (recursive step)
(5) The first L's column (L_{21} in the figure) is computed using the column of the current main block as well as the result of the iterative step

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} P_{11} & 0 \\ 0 & P_{22} \end{pmatrix} \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} \rightarrow$$

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} P_{11}L_{11}U_{11} & P_{11}L_{11}U_{12} \\ P_{22}L_{21}U_{11} & P_{22}L_{21}U_{12} + P_{22}L_{22}U_{22} \end{pmatrix} \rightarrow$$

$$P_{11}L_{11}U_{11} = A,$$
$$U_{12} = L_{11}^{-1}P_{11}^{-1}B,$$
$$P_{22}L_{21} = CU_{11}^{-1},$$
$$P_{22}L_{22}U_{22} = D - P_{22}L_{21}U_{12},$$
$$L_{21} = P_{22}^{-1}CU_{11}^{-1}$$

Figure 7: LU mathematical principles

Finally, Figure 8 shows the code corresponding to the previously presented algorithm.

4 RESULTS

4.1 Computing infrastructure

The execution results presented in this section have been obtained in two different clusters located at the Barcelona Supercomputing Center (BSC).

We have used COMPSs version 2.0 for the initial evaluations, and the same version with the Python persistent workers to test these new features. We have also used Intel®Python 2.7.11 and MKL 2017.

MareNostrum III. This supercomputer was composed by 3056 nodes, each of them with two Intel®SandyBridge-EP E5-2670/1600 20M (8 cores at 2,6 GHz each), main memory that varies from 32 to 128 GB, FDR-10 Infiniband and Gigabit Ethernet network interconnections, and 3 PB of disk storage [8]. It was providing service for researchers from a wide range of different areas, such as life science, earth science and engineering until March 2017.

SSF cluster. This cluster is composed of 8 nodes with two Intel®Xeon®CPU E5-2690 v4 @ 2.60GHz and 128 GB of main memory each (*Xeon nodes*) and 8 nodes with an Intel®Xeon Phi®CPU

```
def lu_blocked(A, MKLProc):
    import numpy as np
    Pres = [[np.matrix(np.zeros((BSIZE, BSIZE)),dtype=float)]
        * MSIZE for _ in range(MSIZE)]
    Lres = [[None] * MSIZE for _ in range(MSIZE)]
    Ures = [[None] * MSIZE for _ in range(MSIZE)]
    for i in range(len(A)):
        for j in range(i+1, len(A)):
            Lres[i][j] = np.matrix(np.zeros((BSIZE, BSIZE)),dtype=float)
            Ures[j][i] = np.matrix(np.zeros((BSIZE, BSIZE)),dtype=float)
    Pres[0][0], Lres[0][0], Ures[0][0] = custom_lu(A[0][0], MKLProc)
    for j in range(1, MSIZE):
        Ures[0][j] = multiply(MKLProc, [1],
            invert_triangular(Lres[0][0], MKLProc, lower = True),
            Pres[0][0], A[0][j])
    for i in range(1, MSIZE):
        for j in range(i, MSIZE):
            for k in range(i, MSIZE):
                mat = invert_triangular(Ures[i - 1][i - 1], MKLProc,
                    lower=False)
                dgemm(-1, A[j][k], multiply(MKLProc, [], A[j][i - 1], mat),
                    Ures[i-1][k], MKLProc)
        Pres[i][i], Lres[i][i], Ures[i][i] = custom_lu(A[i][i], MKLProc)
        for j in range(0, i):
            Lres[i][j] = multiply(MKLProc, [0], Pres[i][i], A[i][j],
                invert_triangular(Ures[j][j], MKLProc, lower = False))
        for j in range(i + 1, MSIZE):
            Ures[i][j] = multiply(MKLProc, [1],
                invert_triangular(Lres[i][i], MKLProc, lower = True),
                Pres[i][i], A[i][j])
    return Pres, Lres, Ures
```

Figure 8: LU factorization main function

7210 @ 1.30GHz and 110 GB of main memory each (*KNL nodes*). The network technology used is OmniPATH. Also, Lustre [6] is used as shared file system.

4.2 Matrix multiplication

For the matrix multiplication example, we have first analyzed the impact of the oversubscription in a single node. The data square matrices considered had 32K by 32K doubles, organized in blocks of 4K and 8K. Larger blocks have not been tested since there is a limit on the serialization size for a single block of 4GB for Python 2.7. This limitation is due to a hardcoded parameter in the *save_bytes* function, present in the class *_pickle.c* of the module in charge of the serializations.

We have instructed the Runtime to schedule always two tasks per NUMA socket, where all task threads are bounded to a single socket as described in Section 2. The number of threads has been varied in each execution from 4 to 32 threads, in such a way that the oversubscription ratio varied from 1 (when 4 threads/task are used) to 8 (when 32 threads/task are used). Figure 9 shows the results of this evaluation. Notice that PyCOMPSs gets the maximum performance when using a level of oversubscription 4 and a block size of 8K. The performance in the best case is around 200 GFLOPS, which represents the 60% of the peak of a MareNostrum III node (332 GFLOPS [7]).

Next, we have executed the same PyCOMPSs code in a variable number of nodes, from 1 to 64 (from 16 to 1024 cores), with an additional node for the master.

Figure 10 shows the performance obtained with matrices of 64K by 64K doubles, organized in blocks of 4K by 4K. The runtime was configured to execute a maximum of four tasks per node, each of

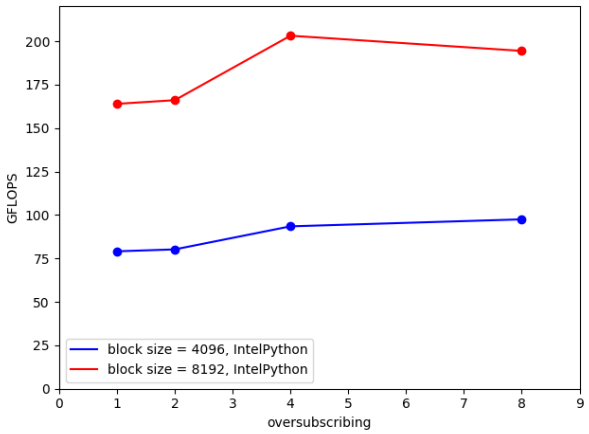

Figure 9: Matrix multiplication evaluation inside one node: impact of oversubscription

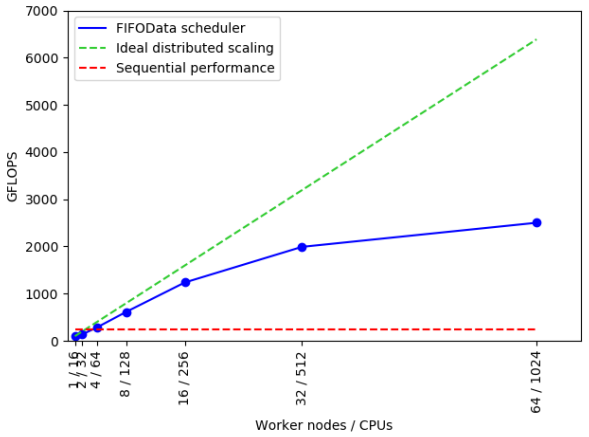

Figure 10: Matrix multiplication performance in a distributed cluster

them using 16 MKL threads, which represent an oversubscription ratio of four. The light blue line labeled *Ideal distributed scaling* is derived by considering an ideal speedup from the performance obtained in one node with PyCOMPSs. The red line *Sequential performance* corresponds to the performance obtained with MKL in one node with 16 threads for a single block of 4K by 4K (251 GFLOPS). We observe a very good speedup until 16 nodes. After that, the efficiency is lower but the system keeps scaling.

The next result worth mentioning concerning the matrix multiplication is the heterogeneous execution in the SSF cluster. Figure 11 shows a post-mortem trace file obtained with Extrae and analyzed with Paraver. More precisely, the execution uses one Xeon node and one KNL node. Blue tasks correspond to initializations and the white ones to block multiplications. Green flags point out beginning and end of tasks. Notice that these many-core architectures allow working with plenty of tasks simultaneously. For instance, this execution multiplies two matrices of 131K x 131K divided into

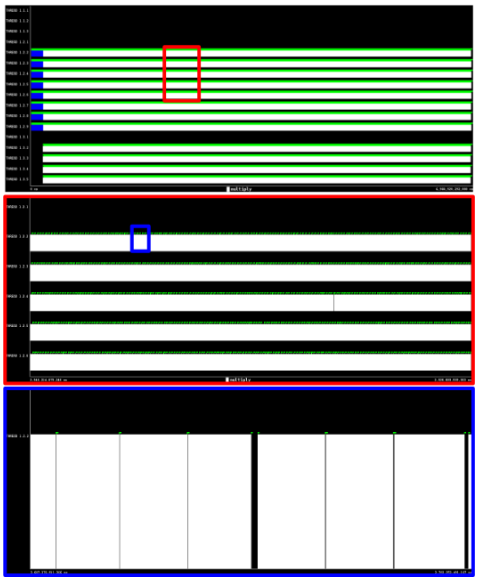

Figure 11: Matrix multiplication heterogeneous execution trace in SSF cluster

blocks of 4K, with 1024 free tasks in average, and a total amount of 32768 tasks; showing the capability of the COMPSs runtime to handle a huge amount of work. Each KNL node executes 4 tasks concurrently and each Xeon node executes 8 tasks at a time.

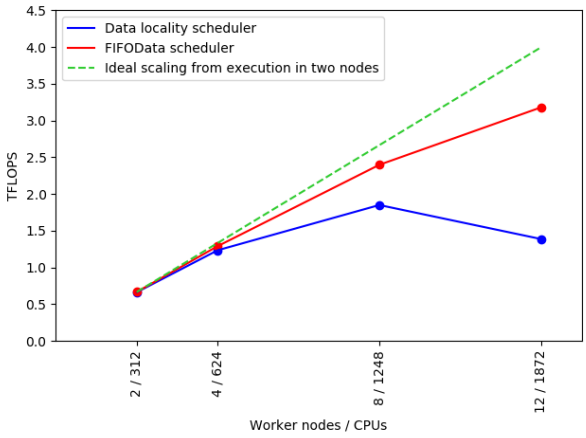

Figure 12: Performance of a 131K x 131K double matrix multiplication heterogeneous execution in the SSF cluster

In this context, and considering that there is not a dedicated node to allocate the master (it runs in a Xeon node that also executes an entire worker), the scalability study has stressed the scheduler. Figure 12 shows how the data locality scheduler degrades much faster than the FIFOData scheduler. This is due to the simplification in the scheduling policy, avoiding a lot of comparisons between the score of the different free tasks. In this case, the users have two options to increase the performance, either to change the scheduler or to dedicate more resources to the Runtime.

4.3 Cholesky factorization

Figure 13 shows the Cholesky performance in MareNostrum III. The matrix size is again 64K by 64K doubles, and the block size 4k by 4k doubles. We have used the same values for the maximum tasks per node and oversubscription than in the previous case (4 tasks per node, 16 threads per task, and an oversubscription ratio of 4).

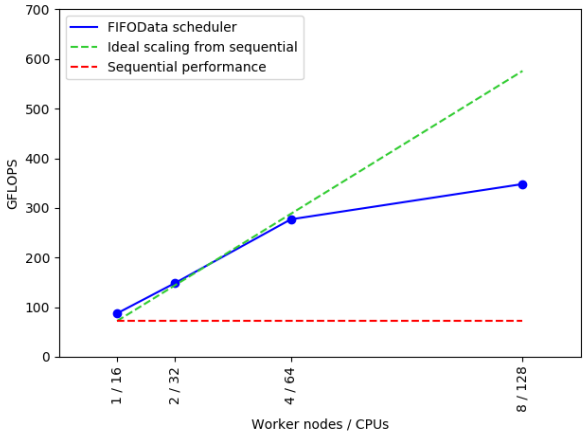

Figure 13: Cholesky performance in MareNostrum III

The red line *Sequential performance* corresponds to the performance obtained with MKL `potrf` with 16 threads for a block of 4K by 4K doubles (72 GFLOPS) in a single MareNostrum node. The green line *Ideal scaling from sequential* is equivalent to the ideal speedup calculated from the previous value.

The chart shows two PyCOMPSs performance results. On the one hand, the blue line corresponds to the performance of PyCOMPSs 2.0. On the other hand, the red line corresponds to the performance of PyCOMPSs 2.0 with the enhancement of the Python persistent workers described in Section 2.4.

Both lines show a similar behavior, with ideal scaling up to four nodes and an increasing degradation from there. When inspecting the post-mortem performance traces, we have observed that the degradation of the performance is due to the morphology of the task-graph. As shown in Figure 14, the maximum available parallelism is already filled with eight nodes. This is a good example of how useful the profiling tools can be when analyzing the code to understand the obtained performance.

Moreover, a significant performance improvement is observed with the new persistent Python worker. This improvement is mainly due to the reduction in the overhead seen in the tasks to start the Python environment and to import the different Python modules used by the tasks.

Figure 15 shows the performance obtained with a heterogeneous execution in the SSF cluster. The block size is 4K x 4K and the matrix size is 130K by 130K. For each execution, half of the nodes are Xeon,

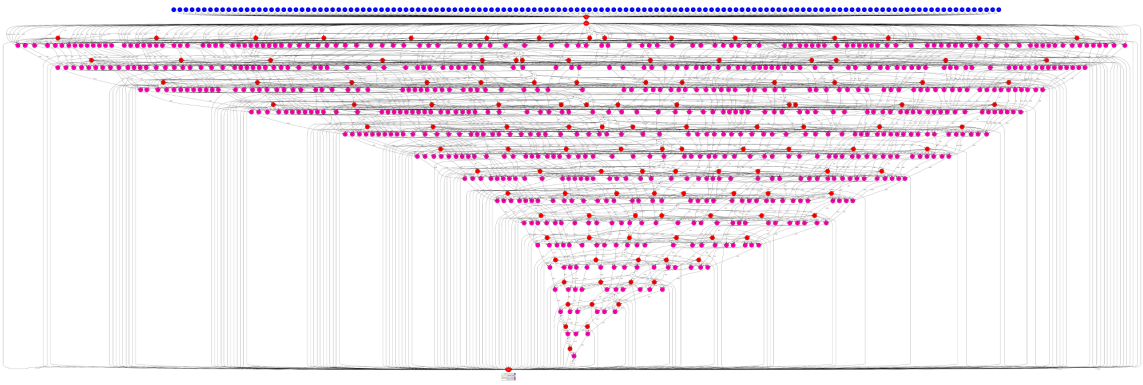

Figure 14: Cholesky dependency graph for a 16 blocks by 16 blocks matrix decomposition

and half are KNL nodes. In this case, only the `gemm` function can run on both architectures. The rest can run only in the Xeon nodes.

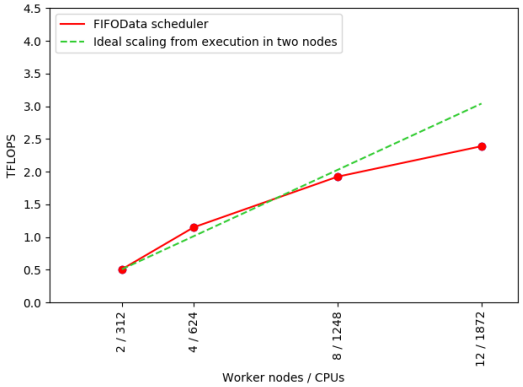

Figure 15: Cholesky performance in SSF

Figure 16 shows the execution trace of the previous performance test with 4 Xeon and 4 KNL nodes. Green segments correspond to initialization tasks, the blue ones to `potrf`, the red ones to `solve_triangular` and the white ones to `gemm`. Yellow lines separate the different nodes. While the nodes with a maximum of 4 tasks running at a time are KNLs, the ones with a maximum of 8 are Xeon nodes. In this execution, the KNL nodes are constrained to execute `gemm` tasks and readers may notice that the scheduler is capable of handling the fact that not all the machines can execute all tasks' types by only sending the tasks where they can run.

4.4 QR factorization

For the QR evaluation, we used a matrix of size 32K by 32K doubles organized in blocks of 4K by 4K doubles. We have evaluated the dense version of the factorization against the memory save version. As described in Section 3.3, the difference between both versions is that the sparse version only allocates those blocks of the auxiliary matrix that are different to the identity or to zero. Additionally, the operations with this identity or zero blocks are not performed (the original block or a zero block is directly returned). To accomplish

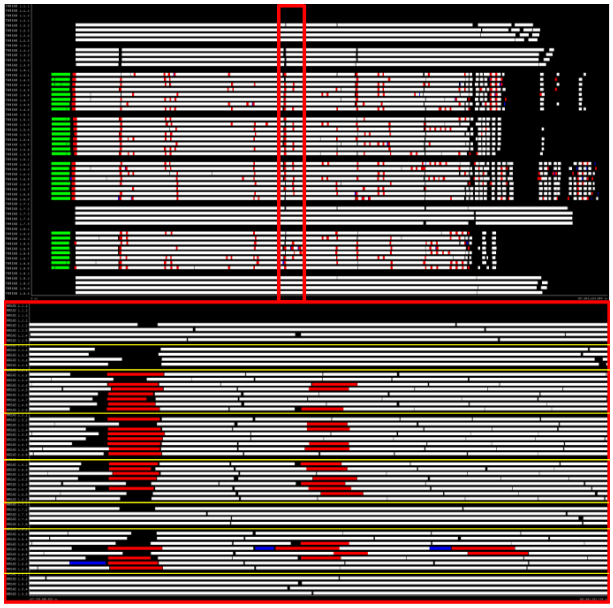

Figure 16: Matrix multiplication heterogeneous execution trace in SSF cluster

this behavior, each element is modeled as a list with an integer to indicate the block type and, if necessary, a NumPy array with its real content.

Figure 17 shows the dependency graph corresponding to a QR decomposition of a 4 blocks x 4 blocks execution, manifesting that the PyCOMPSs runtime is not only able to handle an enormous amount of tasks but also really complex dependency graphs. The real execution (8 blocks x 8 blocks) has a much more complicated graph but is too large to be attached to this paper.

Figure 18 presents the results obtained executing the code in MareNostrum III. As explained in the previous case, the application does not scale well beyond 4 nodes because it reaches the parallelism limit of the application. Nevertheless, the performance until this point is excellent.

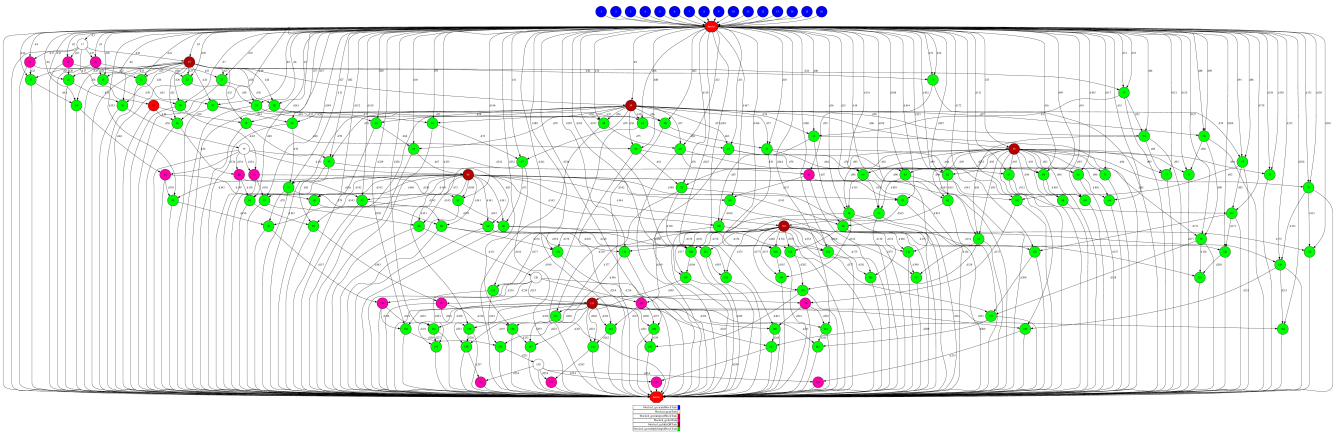

Figure 17: QR dependency graph for a 4 blocks by 4 blocks matrix decomposition

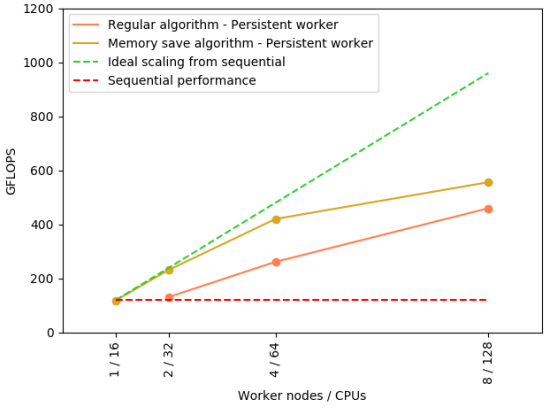

Figure 18: QR performance in MareNostrum III

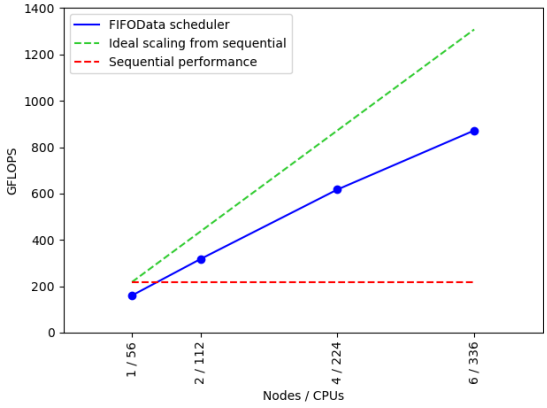

Figure 19: LU performance in SSF cluster with Xeon nodes

4.5 LU factorization

In this case, all the executions have been performed in the SSF cluster using only Xeon nodes. A matrix with 82K x 82K doubles has been factorized, obtaining the matrices P, L and U detailed in Section 3.4.

As shown in Figure 19, the reference execution performance is the maximum achieved with a threaded code executing pure NumPy code, corresponding to a 32K x 32K doubles matrix. The performance obtained with a matrix with 4K x 4K doubles (the block size used in the distributed execution) is 41 GFLOPS, far away from the 218 GFLOPS achieved with the 32K x 32K matrix. This fact suggests that when the matrix is not large enough, not all the resources are fulfilled. Nonetheless, when executing several tasks at the same time, PyCOMPSs take advantage of all the available resources.

5 CONCLUSIONS

PyCOMPSs is a robust programming model that enables the programmer to achieve remarkable performances with clean and simple codes. Based on the sequential version and just by adding some decorators skillfully, the code is ready to run on distributed and heterogeneous clusters thanks to the power of the Runtime that automatically handles all the tasks' scheduling and data transfers. Once the runtime is installed on a given infrastructure, the code parallelized with PyCOMPSs is easily executed.

This paper has shown that good performances can be achieved when calling lower level libraries tuned for each particular architecture. PyCOMPSs allows the users to quickly turn a Python sequential code into a highly portable distributed version. Depending on the code's nature, PyCOMPSs can so play both the distributed programming language and orchestrator roles.

Finally, all the executions have been performed with the data already present in the shared file system. The PyCOMPSs runtime has shown that it is capable of achieving a good performance level when the data is already present in the infrastructure. This fact puts the programming model in the frontier between Big Data and HPC, fulfilling the needs of both environments.

ACKNOWLEDGMENTS

This work has been supported by the Spanish Government (SEV2015-0493), by the Spanish Ministry of Science and Innovation (contract TIN2015-65316-P), by Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272). Javier Conejero postdoctoral contract is co-financed by the Ministry of Economy and Competitiveness under Juan de la Cierva Formación postdoctoral fellowship number FJCI- 2015-24651. Cristian Ramon-Cortes predoctoral contract is financed by the Ministry of Economy and Competitiveness under the contract BES-2016-076791. This work is supported by the Intel-BSC Exascale Lab.

This work has been supported by the European Commission through the Horizon 2020 Research and Innovation program under contract 687584 (TANGO project).

REFERENCES

[1] [n. d.]. *Intel Math Kernel Library. Reference Manual.* Intel Corporation. Santa Clara, USA. ISBN 630813-054US.

[2] (Date of last access: 10th October, 2017). Parallel Processing and Multiprocessing in Python. Web page at https://wiki.python.org/moin/ParallelProcessing. ((Date of last access: 10th October, 2017)).

[3] (Date of last access: 10th October, 2017). Parallel Python Software. Web page at http://www.parallelpython.com. ((Date of last access: 10th October, 2017)).

[4] (Date of last access: 19th December, 2016). Extrae. Web page at https://tools.bsc.es/extrae. ((Date of last access: 19th December, 2016)).

[5] (Date of last access: 19th December, 2016). Paraver: a flexible performance analysis tool. Web page at https://tools.bsc.es/paraver. ((Date of last access: 19th December, 2016)).

[6] (Date of last access: 21st August, 2017). Architecting a High Performance Storage System. Web page at https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/architecting-lustre-storage-white-paper.pdf. ((Date of last access: 21st August, 2017)).

[7] (Date of last access: 21st August, 2017). Intel ®Xeon ®Processor E5-2600 Series. Web page at http://download.intel.com/support/processors/xeon/sb/xeon_E5-2600.pdf. ((Date of last access: 21st August, 2017)).

[8] (Date of last access: 21st August, 2017). MareNostrum III UserâĂŹs Guide. Web page at https://www.bsc.es/support/MareNostrum3-ug.pdf. ((Date of last access: 21st August, 2017)).

[9] (Date of last access: 28th August, 2017). BLAS (Basic Linear Algebra Subprograms). Web page at http://www.netlib.org/blas/. ((Date of last access: 28th August, 2017)).

[10] (Date of last access: 28th August, 2017). Threading Building Blocks (Intel®TBB). Web page at https://www.threadingbuildingblocks.org/. ((Date of last access: 28th August, 2017)).

[11] (Date of last access: 6th October, 2017). PySpark (The Spark Python API). Web page at https://spark.apache.org/docs/latest/api/python/index.html. ((Date of last access: 6th October, 2017)).

[12] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. 2009. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series* 180, 1 (2009), 012037. http://stacks.iop.org/1742-6596/180/i=1/a=012037

[13] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. 1999. *LAPACK Users' guide.* SIAM.

[14] Paolo Bientinesi, Brian Gunter, and Robert A. van de Geijn. 2008. Families of Algorithms Related to the Inversion of a Symmetric Positive Definite Matrix. *ACM Trans. Math. Softw.* 35, 1, Article 3 (July 2008), 22 pages. https://doi.org/10.1145/1377603.1377606

[15] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. 1997. *ScaLAPACK Users' Guide.* Society for Industrial and Applied Mathematics.

[16] Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert van de Geijn. 2008. SuperMatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP '08).* ACM, New York, NY, USA, 123–132. https://doi.org/10.1145/1345206.1345227

[17] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC) (IISWC '09).* IEEE Computer Society, Washington, DC, USA, 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[18] J. Conejero, S. Corella, Rosa M. Badia, and J. Labarta. 2017. Task-based programming in COMPSs to converge from HPC to big data. *International journal of high performance computing applications* (Apr 2017). https://doi.org/10.1177/1094342017701278

[19] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.* 5, 1 (Jan. 1998), 46–55. https://doi.org/10.1109/99.660313

[20] Lisandro DalcÃŋn, Rodrigo Paz, and Mario Storti. 2005. MPI for Python. *J. Parallel and Distrib. Comput.* (2005). http://www.sciencedirect.com/science/article/pii/S0743731505000560

[21] Dask Development Team. 2016. *Dask: Library for dynamic task scheduling.* http://dask.pydata.org

[22] James W. Demmel and Nicholas J. Higham. 1992. Stability of Block Algorithms with Fast Level-3 BLAS. *ACM Trans. Math. Softw.* 18, 3 (Sept. 1992), 274–291. https://doi.org/10.1145/131766.131769

[23] Karim Djemame, Django Armstrong, Richard E. Kavanagh, Jean-Christophe Deprez, Ana Juan Ferrer, David García-Pérez, Rosa M. Badia, Raúl Sirvent, Jorge Ejarque, and Yiannis Georgiou. 2016. TANGO: Transparent heterogeneous hardware Architecture deployment for eNergy Gain in Operation. *CoRR* abs/1603.01407 (2016). http://arxiv.org/abs/1603.01407

[24] Gene H. Golub and Charles F. Van Loan. 1996. *Matrix Computations (3rd Ed.).* Johns Hopkins University Press, Baltimore, MD, USA.

[25] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. 2001. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Softw.* 27, 4 (Dec. 2001), 422–455. https://doi.org/10.1145/504210.504213

[26] Eric Jones, Travis Oliphant, and Pearu Peterson. 2014. {SciPy}: open source scientific tools for {Python}. (2014).

[27] Sheng Liang. 1999. *Java Native Interface: Programmer's Guide and Reference* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[28] Francesc Lordan, Enric Tejedor, Jorge Ejarque, Roger Rafanell, Javier Álvarez, Fabrizio Marozzo, Daniele Lezzi, Raúl Sirvent, Domenico Talia, and Rosa M. Badia. 2014. ServiceSs: An Interoperable Programming Framework for the Cloud. *J. Grid Comput.* 12, 1 (2014), 67–91. https://doi.org/10.1007/s10723-013-9272-5

[29] Hatem Ltaief, Stanimire Tomov, Rajib Nath, Peng Du, and Jack Dongarra. 2011. A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science (VECPAR'10).* Springer-Verlag, Berlin, Heidelberg, 93–101. http://dl.acm.org/citation.cfm?id=1964238.1964251

[30] Wes McKinney. [n. d.]. pandas: a Foundational Python Library for Data Analysis and Statistics. ([n. d.]).

[31] Vincent Pillet et al. 1995. Paraver: A Tool to Visualize and Analyze Parallel Code. *Transputer and occam Developments* (April 1995), 17–32. http://www.bsc.es/paraver - Accessed April, 2012.

[32] Enrique S. Quintana-Ortí and Robert A. van de Geijn. 2008. Updating an LU Factorization with Pivoting. *ACM Trans. Math. Softw.* 35, 2, Article 11 (July 2008), 16 pages. https://doi.org/10.1145/1377612.1377615

[33] Gregorio Quintana-Orti, Enrique S. Quintana-Orti, Ernie Chan, Robert A. van de Geijn, and Field G. Van Zee. 2008. Scheduling of QR Factorization Algorithms on SMP and Multi-Core Architectures. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008) (PDP '08).* IEEE Computer Society, Washington, DC, USA, 301–310. https://doi.org/10.1109/PDP.2008.37

[34] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M Badia, Jordi Torres, Toni Cortes, and Jesús Labarta. 2017. Pycompss: Parallel computational workflows in python. *The International Journal of High Performance Computing Applications* 31, 1 (2017), 66–82.

[35] Guido Van Rossum and Fred L Drake. 2003. *Python language reference manual.* Network Theory.

[36] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. 2011. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science and Engg.* 13, 2 (March 2011), 22–30. https://doi.org/10.1109/MCSE.2011.37

[37] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing.* Berkeley, CA, USA.