Executing linear algebra kernels in heterogeneous distributed infrastructures with PyCOMPSs

Ramon Amela, Cristian Ramon-Cortes, Jorge Ejarque, Javier Conejero, Rosa M. Badia

Oil & Gas Science and Technology - Rev. IFP Energies nouvelles **73**, 47 (2018)

https://doi.org/10.2516/ogst/2018047

Available online at:
www.ogst.ifpenergiesnouvelles.fr

Numerical methods and HPC
A. Anciaux-Sedrakian and Q. H. Tran (Guest editors)

REGULAR ARTICLE OPEN ACCESS

Executing linear algebra kernels in heterogeneous distributed infrastructures with PyCOMPSs

Ramon Amela[1,*], Cristian Ramon-Cortes[1], Jorge Ejarque[1], Javier Conejero[1], and Rosa M. Badia[1,2]

[1] Barcelona Supercomputing Center, Barcelona, Spain
[2] Consejo Superior de Investigaciones Científicas (CSIC), Barcelona, Spain

Received: 24 February 2018 / Accepted: 31 July 2018

Abstract. Python is a popular programming language due to the simplicity of its syntax, while still achieving a good performance even being an interpreted language. The adoption from multiple scientific communities has evolved in the emergence of a large number of libraries and modules, which has helped to put Python on the top of the list of the programming languages [1]. Task-based programming has been proposed in the recent years as an alternative parallel programming model. PyCOMPSs follows such approach for Python, and this paper presents its extensions to combine task-based parallelism and thread-level parallelism. Also, we present how PyCOMPSs has been adapted to support heterogeneous architectures, including Xeon Phi and GPUs. Results obtained with linear algebra benchmarks demonstrate that significant performance can be obtained with a few lines of Python.

1 Introduction

When faced with the selection of a programming language, programmers may take into account aspects such as: syntax simplicity, compilers and tools available, performance attainable, etc. However, other factors such as: language used by the community, libraries specific to the application field, current trends, etc.; can be even more important. In this sense, Python [2] currently fulfills a lot of the requirements: easy to program, good performance trade-off and has a large number of third-party libraries available. Examples of such libraries are NumPy [3] and SciPy [4], that offer vectorized data structures and numerical routines. NumPy is a *de facto* standard when working with tensors in Python due to the high performance achieved, its ease of use and because it automatically maps operations on vectors and matrices to the BLAS [5] and LAPACK [6] functions when present in the system. When a multi-threaded BLAS version is present in the system (using OpenMP [7] or TBB [8]), the operations are automatically parallelized.

Python is an interpreted language and 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.

There are multiple modules that have been developed to provide parallelism in Python, such as the multiprocessing, Parallel Python (PP) and MPI modules. The multiprocessing module provides [9] 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 [10] provides mechanisms for parallel execution of Python codes, with an API with specific functions to specify the number of workers to be used, submits the jobs for execution, gets the results from the workers, etc. Finally, the mpi4py [11] library provides a binding of MPI for Python which allows the programmer to open parallelism both inter-node and intra-node. In all cases, the management of the parallelism is the programmer's responsibility.

PyCOMPSs [12] is a task-based programming model that offers an interface on Python that follows the sequential paradigm. It enables the parallel execution of applications' tasks by means of building, at execution time, a data dependency graph of the tasks that compose an application. The syntax of PyCOMPSs is minimal, using decorators to enable the programmer to identify those methods that are 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

* Corresponding author: ramela@bsc.es

no shared data system is available, synchronizing all activities, acting as an interface with different computing resources such as cloud middlewares, etc. The Runtime also gives 0support to the new container technologies, such as Docker or Singularity [13].

There are other libraries and frameworks that enable Python distributed and multi-threaded executions such as Dask [14] and PySpark [15]. 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 [16] objects. PySpark is a binding to the widely extended framework Spark [17]. A previous paper compares several Big Data algorithms using the native version of both COMPSs[1] and Spark runtimes [18], showing that COMPSs is able to get better or competitive results in comparison to Spark.

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, the support for multiple tasks' versions, and its support to heterogeneous architectures (Xeon Phi and GPUs). This set of extensions allows to achieve good performance with a moderate encoding effort, enabling non 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.

An additional contribution of the 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 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 it easier the development of parallel applications, targeting distributed computing platforms. It relies on the power of its Runtime to exploit the inherent parallelism of the application at execution time by detecting the task calls and the data dependencies between them.

As shown in Figure 1, the PyCOMPSs Runtime allows applications to be executed on top of different infrastructures (such as multi-core machines, grids, clouds or containers) without modifying a single line of the application's code. Thanks to the different connectors, the Runtime is capable of handling all the underlying infrastructure so that the users only define the tasks. It also provides fault-tolerant mechanisms for partial failures (with job resubmission and reschedule when task or resources fail), has a live monitoring tool through a built-in web interface, supports instrumentation using the Extrae [19] tool to generate postmortem traces that can be analyzed with Paraver [20, 21], has an Eclipse IDE, and has pluggable cloud connectors and task schedulers.

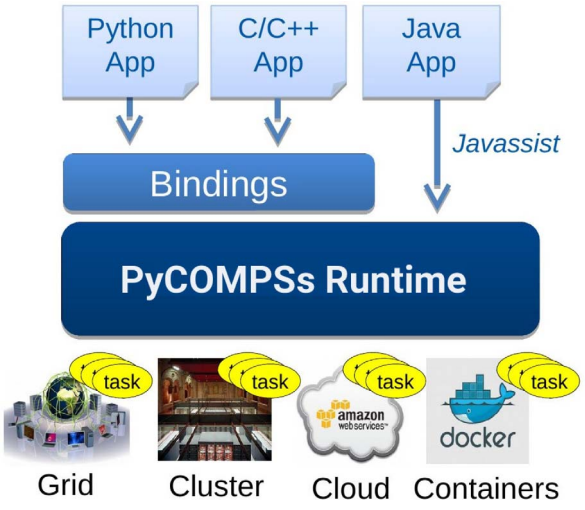


Fig. 1. PyCOMPSs overview.

Moreover, since the PyCOMPSs Runtime is written in Java [22], Python syntax is supported through a binding, PyCOMPSs. This Python binding is supported by a 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 [23].

Regarding the programmability, *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 2 shows an example of a task annotation. The parameter `c` is of type INOUT, and parameters `a`, `b`, and `MKLProc` are set to the default type IN. The directionality tags are used at execution time to derive the data dependencies between tasks and are applied at an object level, taking into account its references to identify when two tasks access the same object. Furthermore, the `priority` tag is a hint for the PyCOMPSs' scheduler that will force to execute the tasks with this tag earlier, always respecting the data dependencies.

Additionally to the `@task` decorator, the `@constraint` decorator can be optionally defined to indicate some task hardware or software requirements. Continuing with the previous example, the task constraint `ComputingUnits` shows to the Runtime how many CPUs 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.

[1] COMPSs is the task-based programming model for which PyCOMPSs is its Python binding.

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

Fig. 2. Sample task annotation.

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

Fig. 3. Sample call to synchronization API.

A tiny synchronization API completes the PyCOMPSs syntax. As shown in Figure 3, the API function `compss_wait_on` waits until all the tasks modifying the `result`'s value are finished and brings the value to the node executing the main program. Once the value is retrieved, the execution of the main program 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 node executing the main program.

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 the 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.

As depicted in Figure 4, the Runtime first builds a task graph adding a new node to it every time a task is invoked in the application's code. The directionality of the parameters is used to detect the data dependencies between the new task and previous ones. Secondly, the scheduler will analyze the generated graph in a particular way to execute all the workload among the available resources. We must highlight that this analysis is highly dependent on the different scheduler implementations, but the Runtime provides information about all the data locations so that they can exploit the *data locality*. Eventually, when a task is scheduled to a given resource, the required objects and files are transferred directly between different memory spaces to guarantee that tasks have available their parameters before execution without passing through the master. Finally, the task is executed in the worker resource and, when specified by the synchronization API, the results are gathered back to the master resource (where the main code is running).

Only concerning the PyCOMPSs binding, when a transfer between different memory spaces is required, the binding serializes and writes the object to disk using the standard library `Pickle`. The transfer between different resources is then delegated to the Runtime.

Finally, the available `Computing Units` that each resource can offer to the Runtime is configurable. More specifically, this allows to oversubscribe the amount of work that a resource can receive; meaning that the Runtime can create more threads than the real amount of CPUs that the resource has.

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 [24]. It has been shown that, by default, Intel®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. It is important to realize that even if a centralized task manager or task scheduler can be a bottleneck in some applications, we are tacking distributed computing platforms and the time required for data transfers can be an issue as well, so we are assuming that tasks should have a given granularity that is able to hide the task manager overhead.

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

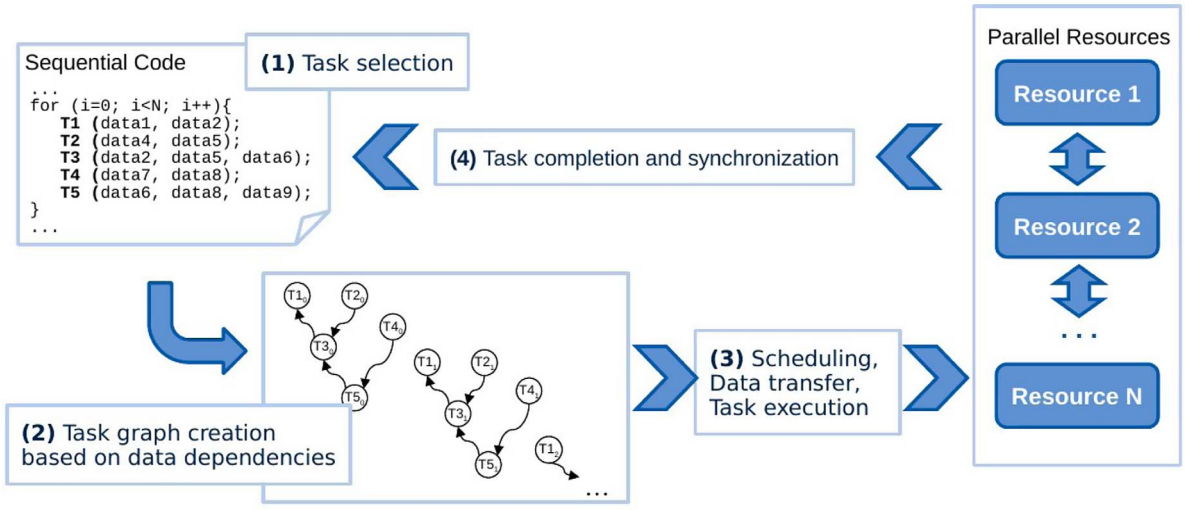


Fig. 4. PyCOMPSs Task life-cycle.

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 users 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 Methods' polymorphism

GPUs are clearly faster than CPUs for some applications. Nevertheless, it is not always easy to decide whether it is better to use one architecture or the other [25]. Also, FPGAs are gaining some momentum. In this context, projects with the primary focus of interest on heterogeneous architectures are arising [26]. Hence, it seems reasonable to think that, in both HPC and Big Data contexts, we are going towards environments with heterogeneous architectures.

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 5 shows an example of polymorphism, 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 polymorphism 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 all 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. In this initial version, data size is not taken into account and we take the hypothesis that each kind of implementation is better or worst in the general case. There are no particularities considered capable to combine both implementations depending on the morphology of the input data, even if it would be really interesting and is can be considered in the future work.

2.6 Profiling

PyCOMPSs generates *postmortem* traces under demand using Extrae [19]. These files can be explored with Paraver [20, 21], 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

```
@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
```

Fig. 5. Version handling with PyCOMPSs.

```
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
```

Fig. 6. Cholesky factorization main function.

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 [27, 28]. Nevertheless, linear algebra algorithms are the base for several fields such as Machine Learning and Computational physics. Even if other good options like ScaLAPACK [29] already do this job, none of them can be called directly from Python. In general, these libraries are coded in `C` or `Fortran`. They are really low level languages that demand a high encoding effort. In return their can achieve really high performances. On the other hand, PyCOMPSs allows the user to have working versions of an algorithm really quick. In addition, the dynamic runtime scheduling empowers the resource efficiency increasing the performance. It is so perfect for prototyping and verifying new concepts and ideas and, as will be shown, can even be competitive under the efficiency point of view.

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 [30]. All the operations performed on the blocks use the best library available for each architecture, either Intel®MKL or `PyCUDA` [31] through `skcuda` [32].

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 Intel®MKL operation. Notice that this behavior happens even when indicated with arithmetic operators.

Figure 2 shows the code used to perform the multiplication task. The computational complexity of this algorithm is widely known: $2S^3$, where S is the side size of the matrix. In this context, it may be easier to note $S = N \times M$ where N is the amount of blocks in a side and M is the side size of a single block.

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 $U = L^t$.

There are two main blocked algorithms to perform a Cholesky decomposition. The right-looking algorithm [33] and the left-looking version [34]. Figure 6 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 and all the potential parallelism is released as soon as possible. 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 Intel®MKL functions, with a given number of threads.

Figure 7 shows the code of the `potrf` task from the Cholesky code. This task has a constraint decorator

```
@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
```

Fig. 7. potrf task in the Cholesky code.

that indicates the number of `ComputingUnits` (CPU's in this case) required to execute the task that, during the experimentation, matches the amount of Intel®MKL threads. Notice that the PyCOMPSs Runtime can be configured to oversubscribe the computing nodes with tasks that involve more threads than the actual available computing cores. This way we can overlap I/O operations when starting/ending tasks with the actual computation, increasing the resource efficiency. The other tasks defined in this example look very similar to the `potrf` one.

On the other hand, this kernel has a GPU version. The PyCOMPSs Runtime automatically handles the GPUs present in a node setting wisely the CUDA visible devices. Figure 8 shows how easy is to take advantage of the accelerators. The polymorphism syntax explained in Section 2 is used to give an alternative version of `dgemm`. The user can consider the presented code as a template and just change the call to *cuBLAS* with the desired kernel. Hence, even if the code can seem a little bit complicated, it allows a user that has never seen a line of *CUDA* to use the GPUs from Python in a distributed context.

Looking at the code and considering the complexity of each call, it is possible to build the Table 1. Hence, the complexity of the algorithm can be computed as shown in Figure 9, where M is the amounts of blocks per side and N the side block size. We can conclude that the algorithm is $O\left(\frac{(MN)^3}{3}\right)$ that is the same complexity of the sequential algorithm.

```
def skcuda_matmul(alpha, A, B, C, beta):
    import os
    import pycuda.autoinit
    import pycuda.gpuarray as gpuarray
    import skcuda.linalg as culinalg
    import skcuda.cublas as cublas
    import ctypes
    ctypes.CDLL("libgomp.so.1", mode=ctypes.RTLD_GLOBAL)
    _libcusolver = ctypes.cdll.LoadLibrary("libcusolver.so")
    culinalg.init()
    from numpy import transpose
    a_gpu = gpuarray.to_gpu(A)
    B = transpose(B)
    b_gpu = gpuarray.to_gpu(B)
    c_gpu = gpuarray.to_gpu(C)
    alpha = np.float32(alpha)
    beta = np.float32(beta)
    cublas_handle = cublas.cublasCreate()
    #In this call we assume A, B, C square
    cublas.cublasDgemm(cublas_handle, "n", "n", A.shape[0],
        B.shape[1], A.shape[1], alpha, b_gpu.gpudata, B.shape[1],
        a_gpu.gpudata, A.shape[1], beta, c_gpu.gpudata, B.shape[0])
    cublas.cublasDestroy(cublas_handle)
    mat_res = c_gpu.get()
    return mat_res

@implement(source_class="cholesky", method="gemm")
@constraint(processors=["ProcessorType":"CPU", "ComputingUnits":1,
    "ProcessorType":"GPU", "ComputingUnits":1])
@task(returns=list)
def gemm_gpu(alpha, A, B, C, beta, MKLProc = 1):
    from gpu_kernels import skcuda_matmul
    res_gpu = skcuda_matmul(alpha, A, B, C, beta)
    return res_gpu
```

Fig. 8. `dgemm` task executed in the GPU.

Table 1. *NumPy* calls done to perform a Cholesky decomposition.

Function	Amount of calls	Complexity
`potrf`	M	$\frac{N^3}{3}$
`Trsm`	$\frac{M\cdot(M+1)}{2}$	$2\cdot N^3$
`gemm`	$M\cdot\frac{M\cdot(M+1)}{2}$	$2\cdot N^3$

$$M\frac{N^3}{3}+\frac{M(M+1)}{2}2N^3+M\frac{M(M-1)}{6}2N^3$$
$$=\frac{MN^3}{3}+M^2N^3+MN^3+\frac{M^3N^3-M^2N^3}{3}\approx\frac{(MN)^3}{3}$$

Fig. 9. Cholesky complexity.

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 [35]. While the Cholesky approach is the well known typical one, in the QR we have to follow a more complicated procedure to achieve a good degree of parallelism. The following steps are followed:

1. Decomposition of the input matrix in blocks

$$A=\begin{pmatrix} M^{0,0} & M^{0,1} & \dots & M^{0,N} \\ M^{1,0} & M^{1,1} & \dots & M^{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ M^{N,0} & M^{N,1} & \dots & M^{N,N} \end{pmatrix}$$

2. QR decomposition of the upper left block

$$M^{i,j}=Q^{i,i}\tilde{R}_i^{i,i}\rightarrow A'=\begin{pmatrix} (Q^{0,0})^t & 0 & \dots & 0 \\ 0 & I & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & I \end{pmatrix}$$

$$\times \begin{pmatrix} M^{0,0} & M^{0,1} & \dots & M^{0,N} \\ M^{1,0} & M^{1,1} & \dots & M^{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ M^{N,0} & M^{N,1} & \dots & M^{N,N} \end{pmatrix}$$

$$= \begin{pmatrix} \tilde{R}^{0,0} & \tilde{M}^{0,1} & \dots & \tilde{M}^{0,N} \\ M^{1,0} & M^{1,1} & \dots & M^{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ M^{N,0} & M^{N,1} & \dots & M^{N,N} \end{pmatrix}$$

3. Change every block of the column for zero blocks

a. Rotation matrix construction through a partial QR decomposition

$$\begin{pmatrix} \tilde{R}_j^{i,i} \\ \tilde{M}^{j+1,i} \end{pmatrix} = \begin{pmatrix} \Sigma^{0,0} & \Gamma^{0,1} \\ \Gamma^{1,0} & \Sigma^{1,1} \end{pmatrix} \begin{pmatrix} \tilde{R}_{j+1}^{i,i} \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} \tilde{R}_{j+1}^{i,i} \\ 0 \end{pmatrix}$$

$$= \begin{pmatrix} \left(\Sigma^{0,0}\right)^t & \left(\Gamma^{1,0}\right)^t \\ \left(\Gamma^{0,1}\right)^t & \left(\Sigma^{1,1}\right)^t \end{pmatrix} \begin{pmatrix} \tilde{R}_j^{i,i} \\ \tilde{M}^{j+1,i} \end{pmatrix}$$

b. Add the rotation matrix to introduce a new zero block in the R matrix

$$A'' = \begin{pmatrix} \left(\Sigma^{0,0}\right)^t & \left(\Gamma^{1,0}\right)^t & \dots & 0 \\ \left(\Gamma^{0,1}\right)^t & \left(\Sigma^{1,1}\right)^t & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & I \end{pmatrix}$$

$$\times \begin{pmatrix} \left(Q^{0,0}\right)^t & 0 & \dots & 0 \\ 0 & I & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & I \end{pmatrix}$$

$$\times \begin{pmatrix} M^{0,0} & M^{0,1} & \dots & M^{0,N} \\ M^{1,0} & M^{1,1} & \dots & M^{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ M^{N,0} & M^{N,1} & \dots & M^{N,N} \end{pmatrix}$$

$$= \begin{pmatrix} \tilde{R}_1^{0,0} & \tilde{M}^{0,1} & \dots & \tilde{M}^{0,N} \\ 0 & \tilde{M}^{1,1} & \dots & \tilde{M}^{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ M^{N,0} & M^{N,1} & \dots & M^{N,N} \end{pmatrix}$$

c. Repeat the previous procedure with all the elements in the column, obtaining the following expression

$$A^{N)} = \tilde{Q}^0 A = \begin{pmatrix} \tilde{R}_N^{0,0} & \tilde{M}_N^{0,1} & \dots & \tilde{M}_N^{0,N} \\ 0 & \tilde{M}^{1,1} & \dots & \tilde{M}^{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & M^{N,1} & \dots & M^{N,N} \end{pmatrix}$$

$$= \begin{pmatrix} R^{0,0} & [M^0] \\ & A_0 \end{pmatrix}$$

4. Return to the point 1, treating the submatrix result of discarding the first row and the first column of the obtained $A^{N)}$ Once all the transformations have been performed, the following expression is obtained:

$$R = \tilde{Q}^N \tilde{Q}^{N-1} \dots \tilde{Q}^1 \tilde{Q}^0 A$$

5. Compute the final Q

$$\hat{Q} = \tilde{Q}^N \tilde{Q}^{N-1} \dots \tilde{Q}^1 \tilde{Q}^0 \rightarrow R = \hat{Q}A \rightarrow A = \hat{Q}^t R \rightarrow Q = \hat{Q}^t$$

Figure 10 shows the QR implementation used in this paper that corresponds to the previous algorithm. 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.

Once again, the complexity can be deduced from the code considering the complexity of each call, building the Table 2. Thus, as demonstrated in Figure 11, the complexity of the algorithm is O $(20\ (NM)^3)$. In this case, the obtained complexity is larger than the sequential one even if it is in the same order. Nevertheless, the parallelism obtained is remarkable so if the main goal is to make the algorithm scalable, the trade-off is worth.

3.4 LU factorization

In this case, an approach without pivoting [36] has been the starting point. The partial pivoting blocked algorithm [37] 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 [38], 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 12 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).

```
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
```

Fig. 10. QR factorization main function.

Table 2. *NumPy* calls to perform a QR decomposition.

Function	Amount of calls	Complexity
`qr`	M	$2 \cdot N^3$
`dot`	$M^2 + \frac{M \cdot (M-1)}{2}$	$2 \cdot N^3$
`littleQR`	$\frac{M \cdot (M-1)}{2}$	$4 \cdot N^3$
`multiplyBlocked`	$\frac{M^2(M-1)}{2} + \frac{M \cdot (2M^2-3M+1)}{6}$	$8 \cdot N^3$

$$
\begin{aligned}
&2MN^3 + 2\left(M^2 + \frac{M \cdot (M-1)}{2}\right)N^3 + 4\left(\frac{M \cdot (M-1)}{2}\right)N^3 \\
&+ 8\left(M^2\frac{(M-1)}{2} + \frac{M \cdot (2M^2-3M+1)}{6}\right)N^3 \\
&= 2MN^3 + 2M^2N^3 + M^2N^3 - MN^3 + 2M^2N^3 - 2MN^3 \\
&+ 8N^3\left(\frac{M^3}{2} - \frac{M^2}{2} + \frac{2M^3}{6} - \frac{3M^2}{6} + \frac{M}{6}\right) \\
&= 12\frac{MN^3}{6} + 12\frac{M^2N^3}{6} + 6\frac{M^2N^3}{6} - 6\frac{MN^3}{6} + 12\frac{M^2N^3}{6} - 12\frac{MN^3}{6} \\
&+ 24\frac{M^3N^3}{6} - 24\frac{M^2N^3}{6} + 16\frac{M^3N^3}{6} - 24\frac{M^2N^3}{6} + 8\frac{MN^3}{6} \\
&= \frac{20M^3N^3}{3} + 3M^2N^3 + \frac{MN^3}{3} \approx \frac{20N^3M^3}{3} = \frac{20}{3}(NM)^3
\end{aligned}
$$

Fig. 11. QR complexity.

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.

Finally, Figure 13 shows the code corresponding to the previously presented algorithm.

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

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

$$
\begin{aligned}
P_{11}L_{11}U_{11} &= A, \\
U_{12} &= L_{11}^{-1}P_{11}^{-1}B, \\
P_{22}L_{21} &= CU_{11}^{-1}, \\
P_{22}L_{22}U_{22} &= D - P_{22}L_{21}U_{12}, \\
L_{21} &= P_{22}^{-1}CU_{11}^{-1}
\end{aligned}
$$

Fig. 12. LU mathematical principles.

```
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
```

Fig. 13. LU factorization main function.

In the previous cases, all the calls had a cubic complexity. On the other hand, this kernel has some calls like `invert_triangular` – which calls to `solve_triangular`, the `MKL` function to invert triangular matrices – with quadratic complexity that have been neglected, building Table 3. In Figure 14 all the terms have been added to compute the total complexity which is $= \left(\frac{8}{3}(M \times N)^3\right)$. Since in sequential a developer would use either a non pivoting algorithm or a pivoting considering the full columns, it makes no sense to compare the complexity of this algorithm against the sequential code. Nevertheless, it is interesting to realize that it is less than a half of the QR asymptotic complexity.

Table 3. *NumPy* calls to perform a LU decomposition

Function	Amount of calls	Complexity
`multiply`	$+\frac{M^3}{3}+\frac{M^2}{2}+\frac{M}{6}+\frac{M^2}{2}+\frac{3\cdot M}{2}+\frac{M^2}{2}-\frac{M}{2}$	$4\cdot N^3$
`dgemm`	$\frac{M^3}{3}+\frac{M^2}{2}+\frac{M}{6}$	$4\cdot N^3$
`custom_lu`	M	$\frac{2\cdot N^3}{3}$

$$\left(M+\frac{M^3}{3}+\frac{M^2}{2}+\frac{M}{6}+\frac{M^2}{2}+\frac{3\cdot M}{2}+\frac{M^2}{2}-\frac{M}{2}\right)4\cdot N^3$$
$$+\left(\frac{M^3}{3}+\frac{M^2}{2}+\frac{M}{6}\right)4\cdot N^3+M\frac{2\cdot N^3}{3}=$$
$$\left(\frac{13}{6}M+\frac{3}{2}M^2+\frac{M^3}{3}\right)4\cdot N^3+\left(\frac{M^3}{3}+\frac{M^2}{2}+\frac{M}{6}\right)4\cdot N^3+M\frac{2\cdot N^3}{3}=$$
$$\left(\frac{7}{3}M+2\cdot M^2+\frac{2}{3}M^3\right)4\cdot N^3\approx\frac{8}{3}(M\cdot N)^3$$

Fig. 14. LU complexity.

4 Results

4.1 Computing infrastructure

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

We have used PyCOMPSs version 2.2 for the initial evaluations, and the same version with the Python persistent workers to test these new features. We have also used Intel®Python 2.7.13 and Intel®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 [39]. It was providing service for researchers from a wide range of different areas, such as life science, earth science and engineering until March 2017.

COBI cluster

This cluster is an Intel® SSF system composed of 8 nodes with two Intel® Xeon® CPU E5-2690 v4 @ 2.60 GHz and 128 GB of main memory each *(Xeon nodes)* and 8 nodes with an Intel® Xeon Phi® CPU 7210 @ 1.30 GHz and 110 GB of main memory each *(KNL nodes)*. The network technology used is Omni-Path. Also, Lustre [40] is used as shared file system.

Minotauro cluster

This cluster is composed of 61 nodes with two Intel® E5649 @ 2.53 GHz, 2 M2090 NVIDIA GPU cards and 24 GB of main memory each with two Infiniband QDR ports and 39 nodes with two Intel® E5-2630 v3 @ 2.4 GHz, 2 K80 NVIDIA GPU cards (2 NVIDIA Kepler GK210 each) and 128 GB of main memory with 1 PCIe 3.0 Mellanox ConnectX-3FDR 56 Gbit port.

4.2 General comments on the evaluation

We consider that Python users will use the intra-node parallelized version of the linear algebra algorithms through MKL (not only a shared memory implementation, but a one optimized for Intel processors) into their applications. Since these kernels can be directly called from Python without any further installation, we have considered them as a really good reference point for *Cholesky, QR,* and *LU.* Hence, instead of comparing the *PyCOMPSs* implementations against other distributed kernels, we have computed the *SpeedUp*'s against the sequential version encoded to be optimal for this kind of processor. Since the sequential version has shared memory and does not perform transfers, we consider that having a good performance when comparing against these executions is enough to assess the good behavior of the executions.

On the other hand, it is important to keep in mind that, for a given matrix size, increasing the block size implies decreasing the number of total blocks. Hence, even if we show that better performances can be obtained when increasing the block size, this fact decreases the number of blocks and thus the maximum parallelism. Moreover, since the input sizes are bigger, each task requires a higher amount of memory. After considering all these factors and trade-offs, we have decided to perform all the tests with blocks of 4K × 4K.

Finally, we would like to highlight the fact that, for most of the executions, we delegate the transfers to the shared file system. Hence, it is difficult to evaluate accurately the transfer time since the shared file systems have lots of data cached in memory and replicated across the cluster. This is the main reason why this study has not been done.

4.3 Matrix multiplication

For the matrix multiplication example, we have first analyzed the impact of the oversubscription in a single node of MareNostrum III. The data square matrices considered had 32K × 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 4 GB 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 in each execution ranged from 4 to 32 threads, in such a way that the oversubscription ratio ranged from 1 (when 4 threads/task are used) to 8 (when 32 threads/task are used). Figure 15 shows the results of this evaluation. Notice that PyCOMPSs gets the maximum performance when using a level of oversubscription of 4 and a block size of 8K. The performance in the best case is around

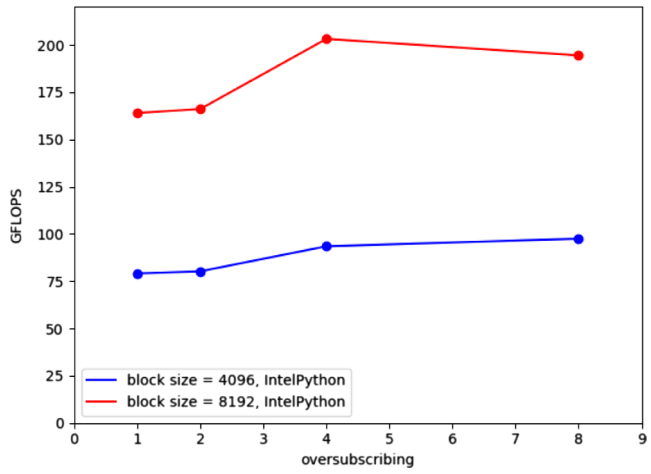


Fig. 15. Matrix multiplication evaluation inside one node: impact of oversubscription.

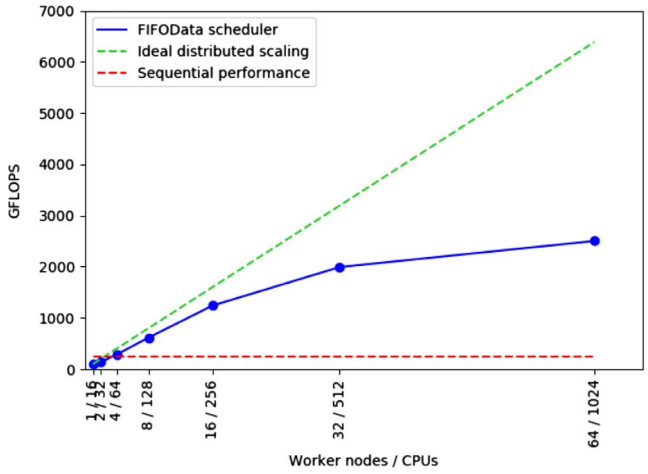


Fig. 16. Matrix multiplication performance in a distributed cluster.

200 GFLOPS, which represents the 60% of the peak of a MareNostrum III node (332 GFLOPS [41]).

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

Figure 16 shows the performance obtained with matrices of 64K × 64K doubles, organized in blocks of 4K × 4K. The Runtime was configured to execute a maximum of four tasks per node, each of them using 16 Intel®MKL threads, which represent an oversubscription ratio of four. The light green 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 Intel®MKL in one node with 16 threads for a single block of 4K × 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 COBI cluster. Figure 17 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 at the left-side of the image correspond to initialization tasks 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 × 131K divided into blocks of 4K, with 1024 free tasks in average, and a total amount of 32 768 tasks; showing the capability of the PyCOMPSs Runtime to handle a huge amount of work. Each KNL node executes four tasks concurrently and each Xeon node executes eight tasks at a time.

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 18 shows how the Data locality scheduler degrades much faster than the FIFOData

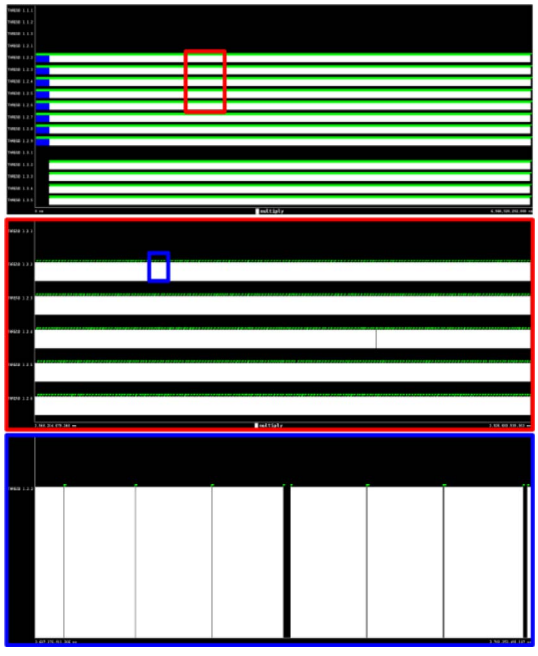


Fig. 17. Matrix multiplication heterogeneous execution trace in the COBI cluster.

scheduler. This is due to the simplification in the second 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.

We have so demonstrated the importance of the choice both of the block size and the scheduler choice. In addition, we show that a good performance on the matrix multiplication can be achieved until 16–32 nodes. Finally, we show that PyCOMPSs Runtime is able to handle different implementations of the same function to take advantage of heterogeneous infrastructures.

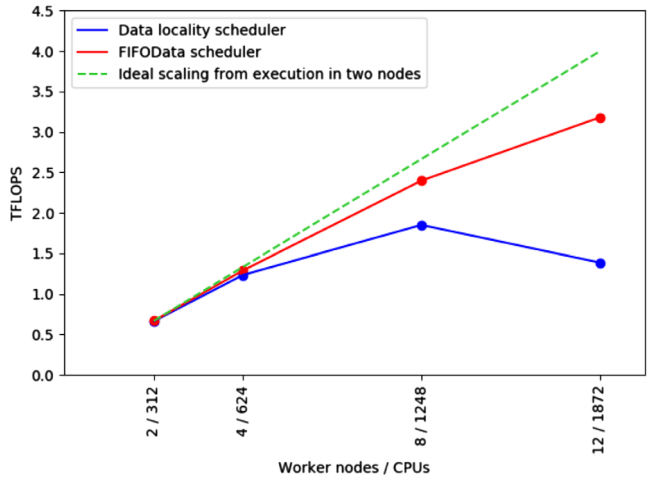


Fig. 18. Performance of a 131K × 131K double matrix multiplication heterogeneous execution in the COBI cluster.

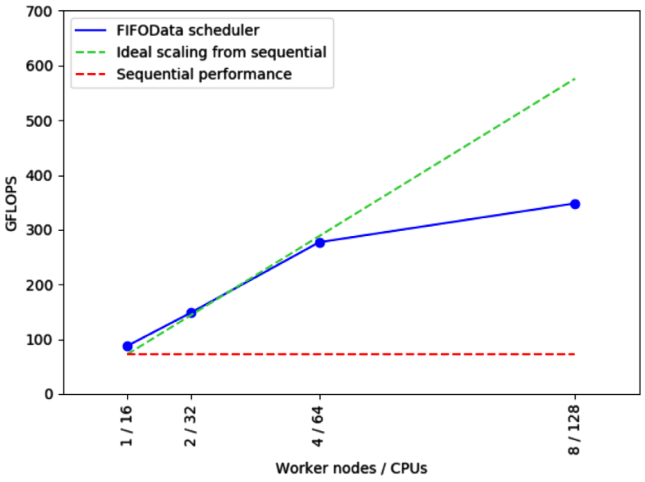


Fig. 19. Cholesky performance in MareNostrum III.

4.4 Cholesky factorization

Figure 19 shows the Cholesky performance in MareNostrum III. The matrix size is again 64K × 64K doubles, and the block size is 4K × 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).

The red line *Sequential performance* corresponds to the performance obtained with Intel®MKL `potrf` with 16 threads for a block of 4K × 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 the performance result with PyCOMPSs 2.2. The line shows 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. Combining the observed results shown in Figures 20 and 21, the maximum available parallelism is already filled with eight nodes. We can conclude this by looking at the inverse triangle form of the DAG and the resource fulfillment of the resources in the trace. This is a good example of how useful the profiling tools can be when analyzing the code to understand the obtained performance.

Figure 22 shows the performance obtained with a heterogeneous execution in the COBI cluster. The block size is 4K × 4K and the matrix size is 130K × 130K. For each execution, half of the nodes are Xeon, 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 20 shows the execution trace of the previous performance test with four Xeon and four 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 four tasks running at a time are KNLs, the ones with a maximum of eight

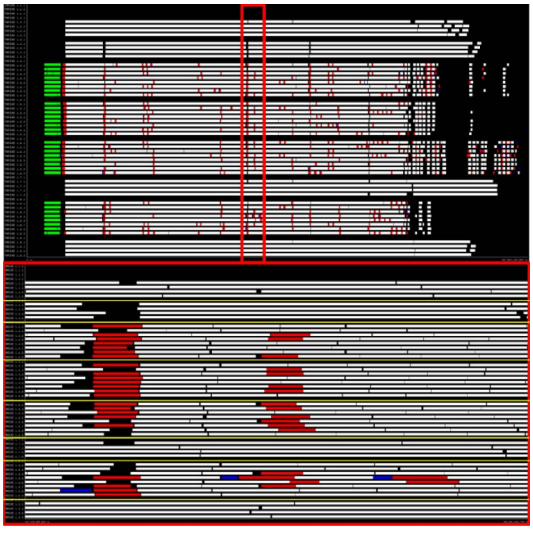


Fig. 20. Cholesky decomposition heterogeneous execution trace in the COBI cluster.

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.

Finally, two executions in the Minotauro cluster have been performed. This is an heterogeneous cluster with two different types of node. It this case, no performance studies have been done. This is mainly due to the fact that considering that each time a GPUs is used, the data has to be copied to its memory and bring back the result. This slows down the execution significantly compared to the maximum theoretical performance. Nevertheless, it is important to realize that the PyCOMPSs Runtime is able to handle a double heterogeneity in this case. On a first level, we have two different kind of nodes in the cluster. On a second level, each one of those nodes has two different kind

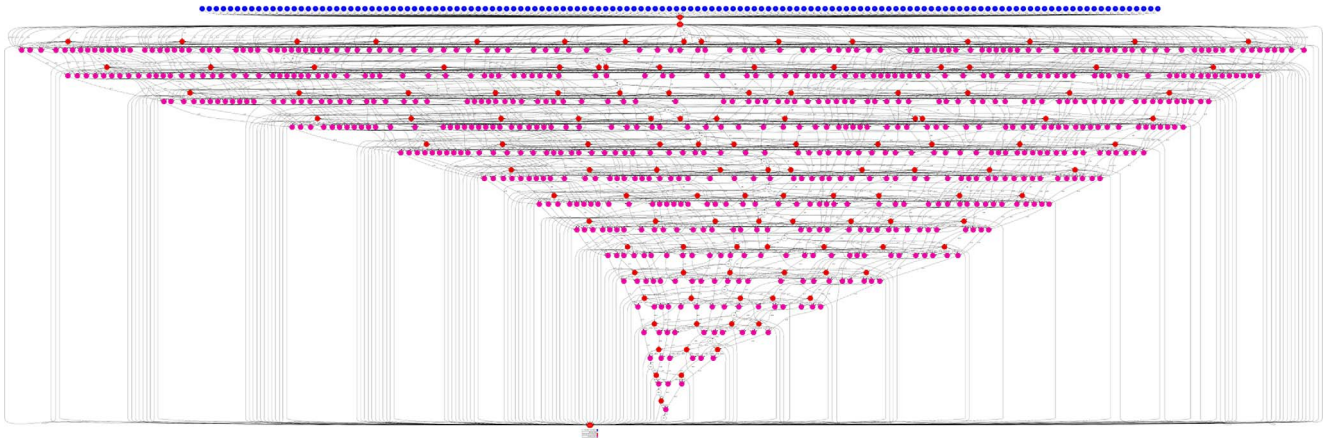


Fig. 21. Cholesky dependency graph for a 16 blocks × 16 blocks matrix decomposition.

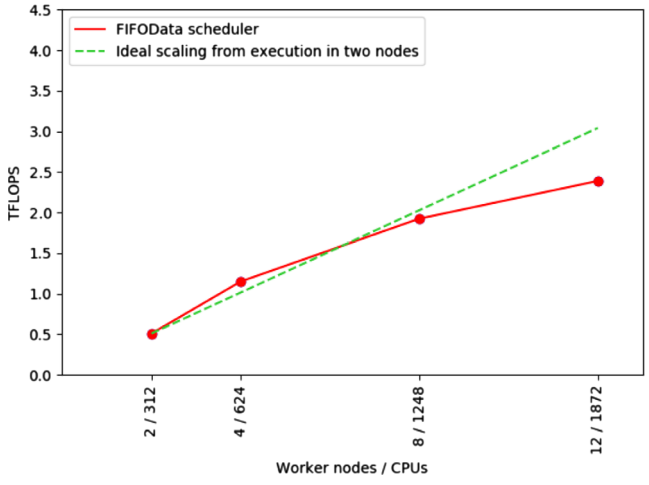
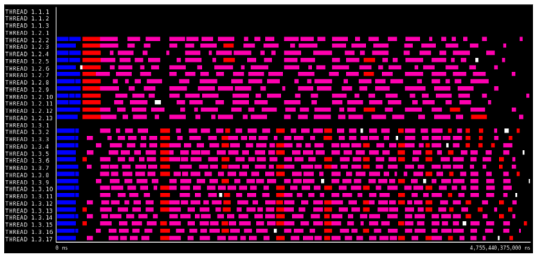
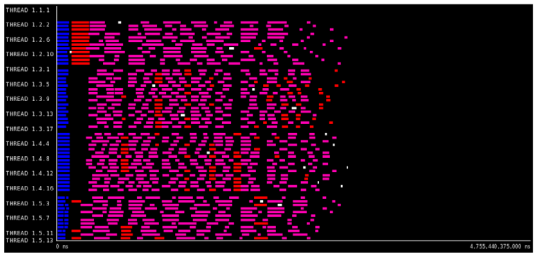


Fig. 22. Cholesky performance in COBI.



(a) 2 execution nodes



(b) 4 execution nodes

Fig. 23. Cholesky factorization: heterogeneous execution trace of a 16 blocks × 16 blocks matrix.

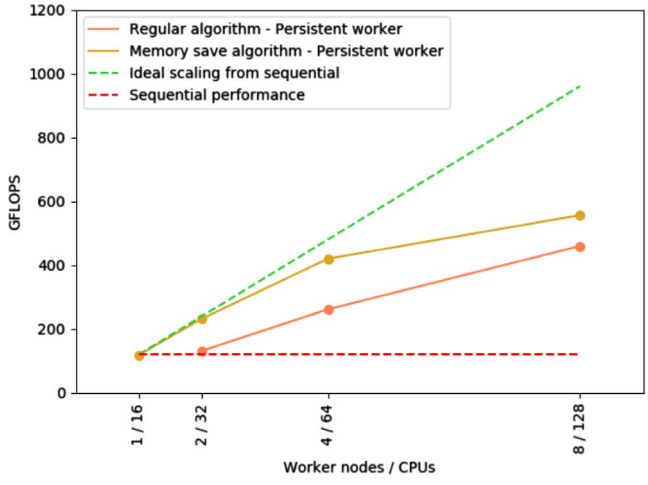


Fig. 24. QR performance in MareNostrum III.

of processors: several CPUs and GPUs. PyCOMPSs Runtime handles successfully four different types of processors in a single execution with a minimum effort for the developer. Figure 23 shows the traces obtained when using two (Fig. 23a) and four nodes (Fig. 23b). In each case, half the nodes are of one type and half of the other. The main interest of these executions is to show that we are able to execute a kernel in an heterogeneous GPU cluster. Still, the performance remains really poor since all the data is copied between the CPU and the GPU memory each time a computation is done. We think that the performance could be increased implementing a cache to handle the GPU memory, avoiding some data transfers. Nevertheless, this remains as an idea for future work.

4.5 QR factorization

For the QR evaluation, we used a matrix of size 32K × 32K doubles organized in blocks of 4K × 4K doubles. We have evaluated the dense version of the factorization against the memory save version. As described in Section 3.3, the

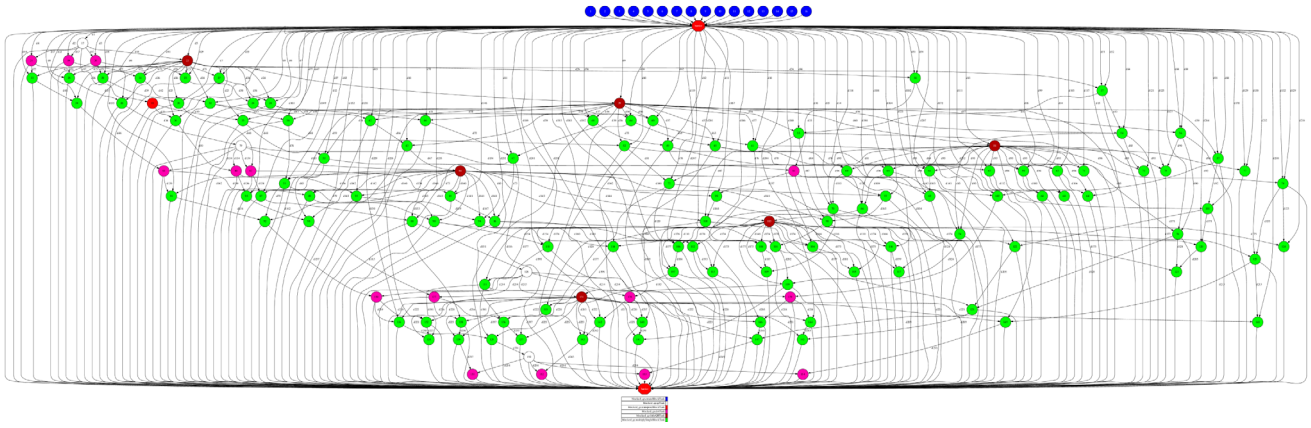


Fig. 25. QR dependency graph for a 4 blocks × 4 blocks matrix decomposition.

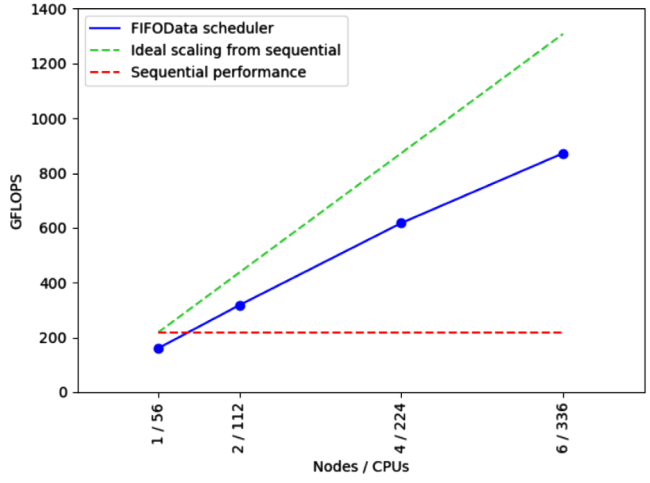


Fig. 26. LU performance in the COBI cluster with Xeon nodes.

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 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 24 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 25 shows the dependency graph corresponding to a QR decomposition of a 4 blocks × 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 × 8 blocks) has a much more complicated graph but is too large to be depicted in this paper.

4.6 LU factorization

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

As shown in Figure 26, the reference execution performance is the maximum achieved with a threaded code executing pure NumPy code, corresponding to a 32K × 32K doubles matrix. The performance obtained with a matrix with 4K × 4K doubles (the block size used in the distributed execution) is 41 GFLOPS, far away from the 218 GFLOPS achieved with the 32K × 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 takes advantage of all the available resources.

Two remarkable facts can be deduced from this experiment. The blocked LU decomposition scales pretty well until a reasonable amount of cores. In addition, the PyCOMPSs Runtime demonstrates its capacity to work with really complicated DAGs. Figure 27 shows a zoom of a dependency graph for a 8 blocks × 8 blocks execution that makes easier to appreciate the complexity handled by the PyCOMPSs Runtime. On the other hand, Figure 28 shows the dependency graph for a 6 blocks × 6 blocks execution.

On the other hand, not only the DAG complexity is important. The amount of tasks that the Runtime can handle is also remarkable. In this case, we have observed that the scheduler choice has a really big impact in the final performance. Figure 29a shows that, in this case, the data locality scheduler is not the best choice. It adds come overhead to compute the best locality when, in fact, all the data is directly available on lustre. When taking into account this fact, the FIFOData scheduler has been considered. Figure 29b shows that the obtained performance is much better. In this execution, 34 288 tasks are executed in 380 seconds in 196 cores. Figure 29c shows that with this amount of tasks and slots available, the Runtime is able to run correctly with tasks that last 0.7–2.0 s. Hence, the throughput is in the microseconds order. This execution can serve as example to better understand PyCOMPSs

Fig. 27. LU dependency graph for a 8 blocks × 8 blocks matrix.

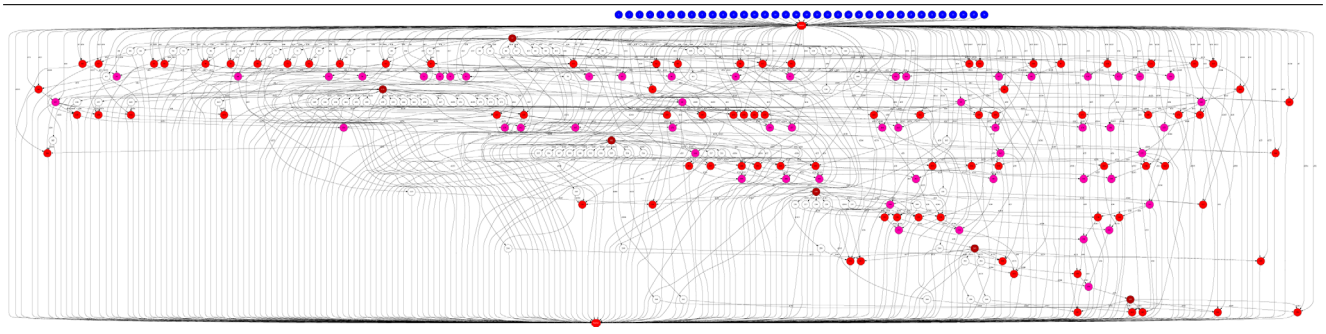


Fig. 28. LU dependency graph for a 6 blocks × 6 blocks matrix decomposition.

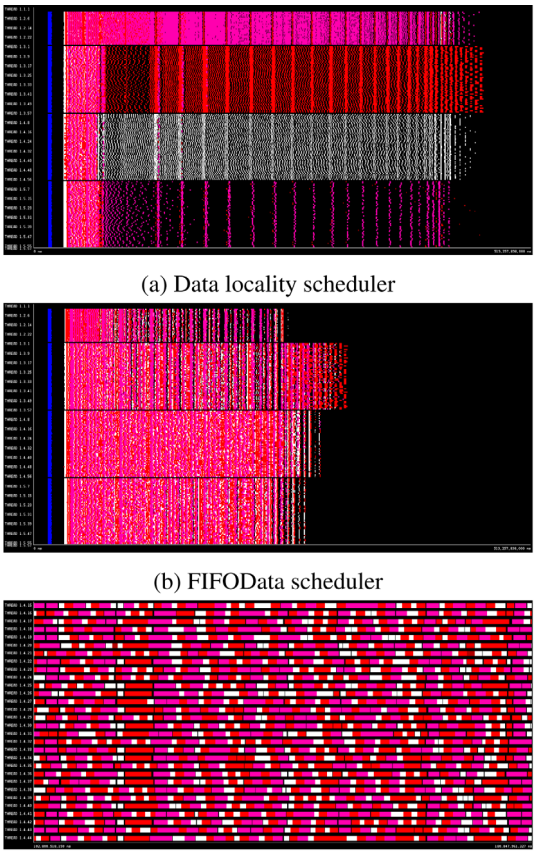
capabilities since is handles a huge amount of tasks not lasting so much.

4.7 Programming complexity

As stated in Section 2, we aim at providing a programming model that makes it easier the development of parallel applications. While it is difficult to evaluate the easiness or complexity of a programming model, we show in this subsection a summary of the lines of code required to program the PyCOMPSs codes described in this paper as a measure for programmability.

These codes are composed of the main algorithm and the tasks' functions. The main algorithm is usually expressed in a method, like the ones shown in Figures 6, 10 or 13, with lengths of 15–30 lines composed of a set of nested loops with calls to the task functions. The task functions are much simpler, each of them of 5–10 lines (like the ones in Figures 2 or 7) and are wrappers to the NumPy calls with the corresponding PyCOMPSs decorators.

Table 4 shows a summary of the lines required to program all the codes presented in this paper. The Main function column is the number of lines in the main algorithm of the codes. The Program column is the total number of lines. The column Decorator corresponds to the number of lines specific for PyCOMPSs, that can be considered the added lines to a sequential version of the code. We can see that from very simple codes, we can obtain performance in distributed and heterogeneous platforms. Readers must consider that the codes contain spaces and blank lines to ease its



(a) Data locality scheduler

(b) FIFOData scheduler

(c) 80 seconds zoom on the FIFOData scheduler execution

Fig. 29. LU decomposition execution trace of a 32 blocks × 32 blocks matrix with 2048 × 2048 blocks.

Table 4. Amount of code lines in each kernel.

Algorithm	Main function	Program	Decorator
`matmul MN3` [1]	7	66	4
`matmul COBI`[2]	7	91	7
`cholesky MN3`[1]	22	136	8
`cholesky COBI`[2]	22	147	11
`cholesky Minotauro`[3]	22	170	11
`QR MN3`[1]	31	233	10
`memorySaveQR MN3`[1]	31	310	14
`LU COBI`[1]	45	155	10

[1] Homogeneous.
[2] `dgemm` versioning Xeon-KNL.
[3] `dgemm` versioning CPU-GPU.

comprehension and that they include the distributed matrix initialization, the auxiliary functions to join the blocked matrix for the result verification, and the timers to compute the elapsed time. Hence, the number of lines can be considered a superior bound (which could be easily reduced) of a complete program that tests the correct behavior of the kernels. Furthermore, we explicitly state the number of lines added as decorators to obtain the distributed version from the blocked one. We highlight that it makes no sense to compare the distributed version against the sequential one since all the kernels can be called in a single MKL line.

5 Conclusion

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 regarding sequential codes 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. In fact, when increasing the size of the matrices, we are able to take advantage of all the architecture, observing super linear *SpeedUp*'s when considering the sequential execution as the reference.

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 Stephen Cass (2017) *The 2017 Top Programming Languages (IEEE Spectrum)*, https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages, accessed: 2018-02-14.

2 Van Rossum G., Drake F.L. (2003) *Python language reference manual. Network Theory.*

3 van der Walt S., Colbert S.C., Varoquaux G. (2011) The NumPy Array: A structure for efficient numerical computation, *Comput. Sci. Eng.* **13**, 2, 22–30, http://dx.doi.org/10.1109/MCSE.2011.37.

4 Jones E, Oliphant E., Peterson P. (2001) *SciPy: Open source scientific tools for Python*, http://www.scipy.org/

5 BLAS (Basic Linear Algebra Subprograms)Web page at http://www.netlib.org/blas/, accessed: 28th August, 2017.

6 Anderson E., Bai Z.J., Bischof C., Susan Blackford L., Demmel J., Dongarra J., Du Croz J., Greenbaum A., Hammarling S., McKenney A., *et al.* (1999) LAPACK Users' guide, *SIAM.*

7 Dagum L., Menon R. (1998) OpenMP: An Industry-Standard API for Shared-Memory Programming, *IEEE Comput. Sci. Eng.* **5**, 1, 46–55, https://doi.org/10.1109/99.660313.

8 Threading Building Blocks (Intel®TBB) (2017) Web page at https://www.threadingbuildingblocks.org/, accessed: 28th August, 2017.

9 Parallel Processing and Multiprocessing in Python (2017) Web page at https://wiki.python.org/moin/ParallelProcessing, accessed: 10th October, 2017.

10 Parallel Python Software (2017) Web page at http://www.parallelpython.com, accessed: 10th October, 2017.

11 Dalcín L., Paz R., Storti M. (2005) MPI for Python, *J. Parallel. Distr. Com.*, http://www.sciencedirect.com/science/article/pii/S0743731505000560.

12 Tejedor E., Becerra Y., Alomar G., Queralt A., Badia R.M., Torres J., Cortes T., Labarta J. (2017) PyCOMPSs: Parallel computational workflows in Python, *Int. J. High Perform. Comput. Appl.* **31**, 1, 66–82.

13 Ramon-Cortes C., Serven A., Ejarque J., Lezzi D., Badia R.M. (2018) Transparent orchestration of task based parallel applications in containers platforms, *J. Grid Computing* **16**, 1, 137–160.

14 Dask Development Team (2016) *Dask: Library for dynamic task scheduling*, http://dask.pydata.org

15 PySpark (The Spark Python API (2017) Web page at https://spark.apache.org/docs/latest/api/python/index.html, accessed: 6th October, 2017.

16 McKinney W. (2011) pandas: a foundational python library for data analysis and statistics, *Python for High Performance and Scientific Computing* 1–9.

17 Zaharia M., Chowdhury M., Franklin M.J., Shenker S., Stoica I. (2010) Spark: Cluster Computing with Working Sets, in: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, Berkeley, CA, USA.*

18 Conejero J., Corella S., Badia Rosa M., Labarta J. (2017) Taskbased programming in COMPSs to converge from HPC to big data, *Int. J. High Perform. Comput. Appl.* **4**, https://doi.org/10.1177/1094342017701278.

19 Extrae Web page at https://tools.bsc.es/extrae, accessed: 19th December, 2016.

20 Pillet V., *et al.* (1995) Paraver: A tool to visualize and analyze parallel code, *Transputer and Occam Developments* **4**, 17–32, http://www.bsc.es/paraver.

21 Paraver: a flexible performance analysis tool Web page at https://tools.bsc.es/paraver, accessed: 19th December, 2016.

22 Lordan F., Tejedor E., Ejarque J., Rafanell R., Álvarez J., Marozzo F., Lezzi D., Sirvent R., Talia D., Badia R.M. (2014) ServiceSs: An Interoperable Programming Framework for the Cloud, *J. Grid Computing* **12**, 1, 67–91, https://doi.org/10.1007/s10723-013-9272-5.

23 Liang S. (1999) *Java Native Interface: Programmer*'s Guide and Reference, 1st edn, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, ISBN 0201325772.

24 Intel Corporation (2015) *Intel Math Kernel Library. Reference Manual*, Intel Corporation, Santa Clara, USA, ISBN 630813-054US.

25 Che S., Boyer M., Boyer M., Meng J., Tarjan D., Sheaffer J.W., Lee S.-H., Skadron K. (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, pp. 44–54, http://dx.doi.org/10.1109/IISWC.2009.5306797.

26 Djemame K., Armstrong D., Kavanagh R.E., Deprez J.-C., Ferrer A.J., García-Pérez D., Badia R.M., Sirvent R., Ejarque J., Georgiou Y. (2016) TANGO: Transparent heterogeneous hardware Architecture deployment for eNergy Gain in Operation, http://arxiv.org/abs/1603.01407.

27 Chan E., Van Zee F.G., Bientinesi P., Quintana-Orti E.S., Quintana-Orti G., van de Geijn R. (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, pp. 123–132, http://doi.acm.org/10.1145/1345206.1345227.

28 Agullo E., Demmel J., Dongarra J., Hadri B., Kurzak J., Langou J., Ltaief H., Luszczek P., Tomov S. (2009) Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects, *J. Phys. Conf. Ser.* **180**, 1, 012–037, http://stacks.iop.org/1742-6596/180/i=1/a=012037.

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

30 Gunnels J.A., Gustavson F.G., Henry G.M., van de Geijn R.A. (2001) FLAME: Formal Linear Algebra Methods Environment, *ACM Trans. Math. Softw.* **27**, 4, 422–455, http://doi.acm.org/10.1145/504210.504213.

31 Klöckner A., Pinto N., Lee Y., Catanzaro B., Ivanov P., Fasih A. (2012) PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation, *Parallel Computing* **38**, 3, 157–174, ISSN 0167-8191. https://doi.org/10.1016/j.parco.2011.09.001.

32 Givon L.E., Unterthiner T., Benjamin Erichson N., Wei Chiang D., Larson E., Pfister L., Dieleman S., Lee G.R., van der Walt S., Menn B., Mihai Moldovan T., Bastien F., Shi X., Schlüter J., Thomas B., Capdevila C., Rubinsteyn A., Forbes M.M., Frelinger J., Klein T., Merry B., Merill N., Pastewka L., Clarkson S., Rader M., Taylor S., Bergeron A., Ukani N.H., Wang F., Zhou Y. (2015) *scikit-cuda 0.5.1: a Python interface to GPU-powered libraries*, http://dx.doi.org/10.5281/zenodo.40565.

33 Bientinesi P., Gunter B., van de Geijn R.A. (2008) Families of Algorithms Related to the Inversion of a Symmetric Positive Definite Matrix, *ACM Trans. Math. Softw.* **35**, 1, 1–3, http://doi.acm.org/10.1145/1377603.1377606.

34 Ltaief H., Tomov S., Nath R., Du P., Dongarra J. (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, pp. 93–101, http://dl.acm.org/citation.cfm?id=1964238.1964251

35 Quintana-Orti G., Quintana-Orti E.S., Chan E., van de Geijn R.A., Van Zee F.G. (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, pp. 301–310, http://dx.doi.org/10.1109/PDP.2008.37..

36 Golub Gene H., Van Loan C.F. (1996) *Matrix Computations*, 3rd edn., Johns Hopkins University Press, Baltimore, MD, USA, ISBN 0-8018-5414-8.

37 Quintana-Ortí E.S., van de Geijn R.A. (2008) Updating an LU factorization with pivoting, *ACM Trans. Math. Softw.* **35**, 2, 1–11, http://doi.acm.org/10.1145/1377612.1377615.

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

39 MareNostrum III User's GuideWeb page at https://www.bsc.es/support/MareNostrum3-ug.pdf, accessed: 21st August, 2017.

40 Architecting a High Performance Storage SystemWeb page at https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/architecting-lustre-storage-white-paper.pdf, accessed: 21st August, 2017.

41 Intel ®Xeon ®Processor E5-2600 SeriesWeb page at http://download.intel.com/support/processors/xeon/sb/xeon_E5-2600.pdf, accessed: 21st August, 2017.