AutoParallel: Automatic parallelisation and distributed execution of affine loop nests in Python

Cristian Ramon-Cortes, Ramon Amela, Jorge Ejarque, Philippe Clauss, Rosa Badia

AutoParallel: Automatic parallelisation and distributed execution of affine loop nests in Python

The International Journal of High Performance Computing Applications
XX(X):1–14

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Cristian Ramon-Cortes[1], Ramon Amela[1], Jorge Ejarque[1], Philippe Clauss[2], Rosa M. Badia[1]

Abstract

The last improvements in programming languages and models have focused on simplicity and abstraction; leading Python to the top of the list of the programming languages. However, there is still room for improvement when preventing users from dealing directly with distributed and parallel computing issues. This paper proposes and evaluates AutoParallel, a Python module to automatically find an appropriate task-based parallelisation of affine loop nests and execute them in parallel in a distributed computing infrastructure. It is based on sequential programming and contains one single annotation (in the form of a Python decorator) so that anyone with intermediate-level programming skills can scale up an application to hundreds of cores.

The evaluation demonstrates that AutoParallel goes one step further in easing the development of distributed applications. On the one hand, the programmability evaluation highlights the benefits of using a single Python decorator instead of manually annotating each task and its parameters or, even worse, having to develop the parallel code explicitly (e.g., using OpenMP, MPI). On the other hand, the performance evaluation demonstrates that AutoParallel is capable of automatically generating task-based workflows from sequential Python code while achieving the same performances than manually taskified versions of established state-of-the-art algorithms (i.e., Cholesky, LU, and QR decompositions). Finally, AutoParallel is also capable of automatically building data blocks to increase the tasks' granularity; freeing the user from creating the data chunks, and re-designing the algorithm. For advanced users, we believe that this feature can be useful as a baseline to design blocked algorithms.

Keywords

Automatic Parallelisation, Distributed Computing, Programming Models

1 Introduction

Computer simulations have become more and more crucial to both theoretical and experimental studies in many different fields, such as structural mechanics, chemistry, biology, genetics, and even sociology. Several years ago, small simulations (with up to several cores or even several nodes within the same grid) were enough to fulfil the scientific community requirements and thus, the experts of each field were capable of programming and running them. However, nowadays, simulations requiring hundreds or thousands of cores are widely used and, to this point, efficiently programming them becomes a challenge even for computer scientists. On the one hand, interdisciplinary teams have become popular, with field experts and computer scientists joining their forces together to keep their research at the forefront. On the other hand, programming languages have made a considerable effort to ease programmability while maintaining acceptable performance. In this sense, Python van Rossum and Drake (2011) has risen to the top language for nonexperts Cass (2019), being easy to program while maintaining a good performance trade-off and having a large number of third-party libraries available. Similarly, Go Google (2019) has also gained some momentum thanks to its portability, reliability, and ease of concurrent programming, although it is still in its early stages.

Even if some great efforts have been accomplished for programming frameworks to ease the development of distributed applications, we go one step further with AutoParallel: a Python module to automatically parallelise applications and execute them in distributed environments. Our philosophy is to ease the development of parallel and distributed applications so that anyone with intermediate-level programming skills can scale up an application to hundreds of cores. In this sense, AutoParallel is based on sequential programming and only requires a single Python decorator that frees the user from manually taskifying the original code. Internally, it relies on Pluto Bondhugula and et al. (2008b) to parallelise affine loop nests and taskifies the obtained code so that PyCOMPSs can distributedly execute it using any underlying infrastructure (clusters, clouds, and containers). Moreover, to avoid single instruction tasks, AutoParallel can also increase the tasks' granularity by automatically building data blocks (chunks).

[1]Barcelona Supercomputing Center (BSC), Spain
[2]ICube Lab. - Université de Strasbourg, Strasbourg, France

Corresponding author:
Cristian Ramon-Cortes, Barcelona Supercomputing Center (BSC), Carrer Jordi Girona 29, 08034 Barcelona, Spain
Email: cristian.ramoncortes@bsc.es

The rest of the paper is organised as follows. Section 2 describes the state of the art. Section 3 presents PyCOMPSs and Pluto, and Section 4 describes the AutoParallel's architecture. Next, Section 5 evaluates its programmability, Section 6 presents its performance results, and Section 7 analyses the automatic building of data blocks. Finally, Section 8 concludes the paper and gives some guidelines for future work.

2 State of the Art

Nowadays, simulations are run in distributed environments and, although Python has become a reference programming language, there is still much work to do to ease parallel and distributed computing issues. In this concern, Python can provide parallelism at three levels. First, parallelism can be achieved internally through many libraries such as NumPy var der Walt et al. (2011) and SciPy Jones et al. (2001–), which offer vectorised data structures and numerical routines that automatically map operations on vectors and matrices to the BLAS University of Tennesse. Oak Ridge National Laboratory. Numerical Algorithms Group Ltd. (2017) and LAPACK Anderson and et al. (1999) functions; executing the multi-threaded BLAS version (using OpenMP Dagum and Menon (1998) or TBB Intel (2019)) when present in the system. Notice that, although parallelism is completely transparent for the application user, parallel libraries only benefit from intra-node parallelism, while our solution aims for distributed computing. Moreover, NumPy offers vectorised data structures and operations in a transparent way to prevent users from defining loops to handle NumPy values directly. In contrast, our solution requires a Python decorator on top of a method containing affine loop nests. Thus, to parallelise a vector operation, the users must explicitly define the loop nests to apply an operation to all the vector elements. However, in our previous work, Amela and et al. (2017) and Amela and et al. (2018), we have demonstrated the benefits from combining inter- and intra-node parallelism using PyCOMPSs Tejedor and et al. (2017) and NumPy. Similarly, some NumPy extensions can be integrated with PyCOMPSs to boost the intra-node performance. For instance, NumExpr Cooke et al. (2020) can be used to optimise the computation of numerical expressions and, as detailed in Barcelona Supercomputing Center (BSC) (2020), the Numba Lam et al. (2015); Anaconda (2020) compiler annotations can be combined with the PyCOMPSs programming model.

Secondly, many modules can explicitly provide parallelism. The multiprocessing module Python Software Fundation (2019) provides support for the spawning of processes in SMP machines using an API similar to the threading module, with explicit calls for creating processes. In addition, the Parallel Python (PP) module Vitalii Vanovschi (2019) provides mechanisms for parallel execution of Python codes, with an API that includes specific functions for specifying the number of workers to be used, submitting the jobs for execution, getting the results from the workers, etc. Also, the mpi4py Dalcín et al. (2005) library provides a binding of MPI for Python which allows the programmer to handle parallelism both inter-node and intra-node. However, in all cases, the burden of parallelism specific issues is assigned to the programmer.

Third, other libraries and frameworks enable Python distributed and multi-threaded computations such as PyCOMPSs Tejedor and et al. (2017), Dask Dask Development Team (2016), PySpark Apache Software Fundation (2019), and Pydron Müller and et al. (2014). PyCOMPSs is a task-based programming model that targets sequential programming and provides a set of decorators to enable the programmer to identify methods as tasks and a small synchronisation API. Its runtime exploits the inherent parallelism of the applications by building, at execution time, a data dependency graph of the tasks and executing them using a distributed parallel platform (clusters, clouds, and containers). Further details are available on Section 3 since our solution uses this framework. On the other hand, Dask is a native Python library that allows the creation and distributed execution of Directed Acyclic Graphs (DAG) of a set of operations on NumPy and pandas McKinney (2011) objects. Also, PySpark is a binding to the widely extended framework Spark Zaharia and et al. (2010). Finally, Pydron Müller and et al. (2014) is a semi-automatic parallelisation Python module that transparently translates the application into a data-flow graph which can be distributed across clusters or clouds. It offers a `@schedule` decorator to automatically parallelise calls to methods that are annotated with the `@functional` decorator. In comparison to PyCOMPSs, Pydron's `@functional` decorator is equivalent to PyCOMPSs' `@task` decorator. However, Pydron analyses the abstract syntax tree of the function's code to build the data dependency graph while PyCOMPSs requires extra parameter annotations inside the `@task` decorator. Moreover, the `@schedule` decorator enables users to activate and deactivate the parallelisation of `@functional` methods, while PyCOMPSs parallelises any call to a `@task` method.

3 Technical Background

This section provides a general overview of the satellite frameworks that directly interact with the AutoParallel module: PyCOMPSs and Pluto. It also highlights some of their features that are crucial for their integration.

3.1 PyCOMPSs

COMPSs Badia and et al. (2015); Lordan and et al. (2014) is a task-based programming model that aims to ease the development of parallel applications, targeting distributed computing platforms. It relies on its runtime to exploit the inherent parallelism of the application at execution time by detecting the task calls and the data dependencies between them.

The COMPSs runtime natively supports Java applications but also provides bindings for Python and C/C++. Precisely, the Python binding is known as PyCOMPSs. All the bindings are supported through a *binding-commons* layer which focuses on enabling the functionalities of the runtime to other languages. It is written in C and has been designed as an API with a set of defined functions to communicate with the runtime through the JNI Liang (1999).

As shown in Figure 1, the COMPSs runtime allows applications to be executed on top of different infrastructures (such as multi-core machines, grids, clouds or containers Ramon-Cortes and et al. (2018)) without modifying a single line of the application code. Thanks to the different connectors, the runtime is capable of handling all the underlying infrastructure so that the user only defines the tasks. It also provides fault-tolerant mechanisms for partial failures (with job re-submission and reschedule when tasks or resources fail), has a live monitoring tool through a built-in web interface, supports instrumentation using the Extrae Barcelona Supercomputing Center (BSC) (2019b) tool to generate post-mortem traces that can be analysed with Paraver Barcelona Supercomputing Center (BSC) (2019d), has an Eclipse IDE, and has pluggable cloud connectors and task schedulers.

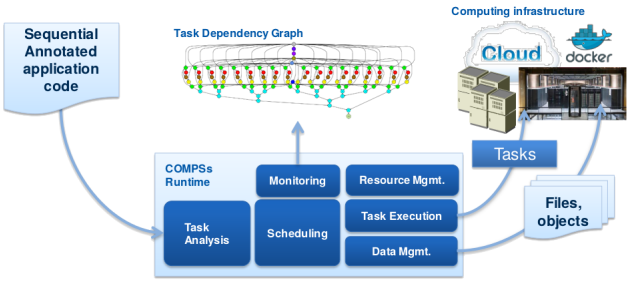

Figure 1. COMPSs overview.

Additionally, the programming model is based on sequential programming which means that users do not need to deal with any parallelisation and distribution issue such as thread creation, synchronisation, data distribution, messaging or fault-tolerance. Instead, application developers only select which methods must be considered as tasks, and the runtime spawns them asynchronously on a set of resources instead of executing them locally and sequentially.

3.1.1 PyCOMPSs Programming Model

Regarding programmability, tasks are identified by inserting annotations in the form of Python decorators. These annotations are inserted at method level and indicate that invocations to a given method should 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).

Listing 1 shows an example of a task annotation. The parameter `c` has direction INOUT, and parameters `a` and `b` are set to the default direction 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.

```
1  @constraint(ComputingUnits="$CUS")
2  @task(c=INOUT)
3  def multiply(a, b, c):
4      c += a * b
```

Listing 1: Sample task annotation.

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` tells 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 the processor architecture, memory size, disk storage, operating system or available libraries.

A tiny synchronisation API completes the PyCOMPSs syntax. As shown in Listing 2, the API function `compss_wait_on` waits for the completion of all the tasks modifying the `result`'s value and brings the final value to the node executing the main program. Then, the execution of the main program is resumed. Given that PyCOMPSs is used mostly in distributed environments, synchronisation implies a data transfer from remote storage or memory space to the node executing the main program.

```
1  for block in data:
2      partial_res = wordcount_task(block)
3      reduce_task(result, partial_res)
4  final_result = compss_wait_on(result)
```

Listing 2: Sample call to synchronisation API.

3.2 Pluto

Many compute-intensive scientific applications spend most of their execution time running nested loops. The Polyhedral Model Cohen and et al. (2005) provides a powerful mathematical abstraction to analyse and transform loop nests in which the data access functions and loop bounds are affine combinations (linear combinations with a constant) of the enclosing loop iterators and parameters. This model represents the instances of the loop nests' statements as integer points inside a polyhedron, where inter and intra-statement dependencies are characterised as a dependency polyhedron. Combining this representation with Linear Algebra and Integer Linear Programming, it is possible to reason about the correctness of a sequence of complex optimising and parallelising loop transformations.

Pluto Bondhugula (2017); Bondhugula and et al. (2008b) is an automatic parallelisation tool based on the Polyhedral model to optimise arbitrarily nested loop sequences with affine dependencies. At compile time, it analyses C source code to optimise and parallelise affine loop-nests and automatically generate OpenMP C parallel code for multi-cores. Although the tool is fully automatic, many options are available to tune tile sizes, unroll factors, and outer loop fusion structure.

As shown in Figure 2, Pluto internally translates the source code to an intermediate OpenScop Bastoul (2011) representation using Clan Bastoul and et al. (2003). Next, it relies on the Polyhedral Model to find affine transformations for coarse-grained parallelism, data locality, and efficient tiling. Finally, Pluto generates the OpenMP C code from the OpenScop representation using CLooG Bastoul (2004). We must highlight that the generated code is also optimised for data locality and made amenable to auto-vectorisation.

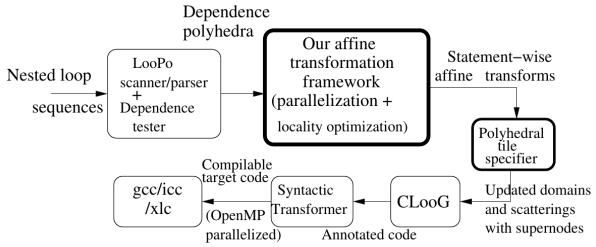

Figure 2. Pluto source-to-source transformation. Source: Bondhugula and et al. (2008a).

3.2.1 Loop Tiling

Among many other options, Pluto can tile code by specifying the `--tile` option. In general terms, as shown in Listing 3, tiling a loop of given size `N` results in a division of the loop in `N/T` repeatable parts of size `T`. For instance, this is suitable when fitting loops into the L1 or L2 caches or, in the context of this paper, when building the data blocks to increase the tasks' granularity.

```
1  # Original loop          # Tiled loop
2  for i in range(N):       for i in range(N/T):
3      print(i)                 for t in range(T):
4                                   print(i*T + t)
```

Listing 3: Example of loop tiling.

Along with this option, users can let Pluto set the tile sizes automatically using a rough heuristic, or manually define them in a `tile.sizes` file. This file must contain one tile size on each line and as many tile sizes as the loop nest depth.

In the context of parallel applications, tile sizes must be fine-tuned for each application so that they maximise locality while making sure there are enough tiles to keep all cores busy.

4 Architecture

The framework proposed in this paper eases the development of distributed applications by letting users program their application in a standard sequential fashion. It is developed on top of PyCOMPSs and Pluto. When automatically parallelising sequential applications, users must only insert an annotation on top of the potentially parallel functions to activate the AutoParallel module. Next, the application can be launched using PyCOMPSs.

Following a similar approach than PyCOMPSs, we have included a new decorator `@parallel` to specify which methods should be automatically parallelised at runtime. Notice that, since PLUTO and the Polyhedral Model can only be applied to affine loops, the functions using this decorator must contain loop nests in which the data access functions and loop bounds are affine combinations (linear combinations with a constant) of the enclosing loop iterators and parameters. Otherwise, the source code will remain intact.

As shown in Figure 3, the AutoParallel Module analyses the user code searching for `@parallel` annotations. Essentially, when found, the module calls Pluto to generate its parallelisation and substitutes the user code by a newly generated code. Once all annotations have been processed, the new tasks are registered into PyCOMPSs, and the execution continues as a regular PyCOMPSs application (as described in Section 3.1). Finally, when the application has ended, the generated code is stored in a (`_autogen.py`) file and the user code is restored.

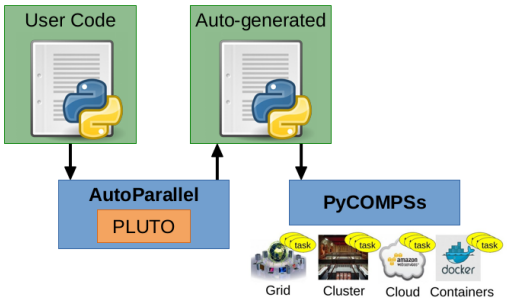

Figure 3. Overview of the AutoParallel Module.

4.1 AutoParallel Module

The internals of the AutoParallel module are quite complex (more than 5.000 lines of code) because it automatically re-writes the python's AST representation of the user code at execution time, while interacting with C/C++ libraries in an intermediate format (OpenScop). This section only provides a first approach to AutoParallel by detailing its five main components. Complete code and documentation can be found in Ramon-Cortes (2019b). For the sake of clarity, Figure 4 shows the relationship between them and their expected inputs and outputs.

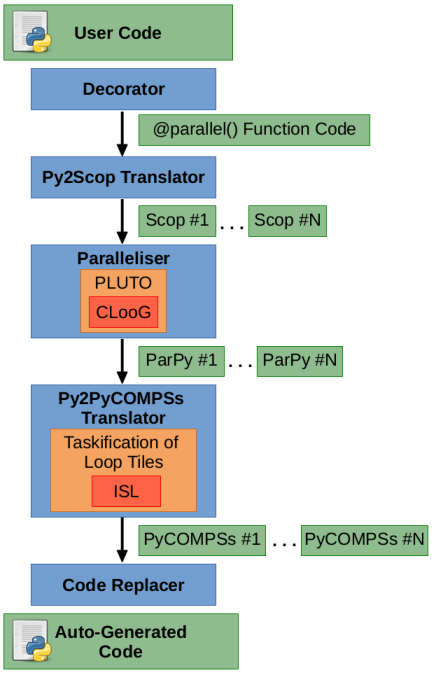

Figure 4. Internals of the AutoParallel Module.

- **Decorator** Implements the `@parallel` decorator to detect functions that the user has marked as potentially parallel.

- **Python To OpenScop Translator** For each affine loop nest detected in the user function, builds a Python Scop object representing it that can be bulked into an OpenScop format file.
- **Paralleliser** Returns the Python code resulting from parallelising an OpenScop file. Since Python does not have any standard regarding parallel annotations, the parallel loops are annotated using comments with OpenMP syntax.
- **Python to PyCOMPSs Translator** Converts an annotated Python code into a PyCOMPSs application by inserting the necessary task annotations and data synchronisations. This component can also build data blocks from loop tiles and taskify them if enabled by the user (see Section 4.2 for more details).
- **Code Replacer** Replaces each loop nest in the initial user code by the auto-generated code so that PyCOMPSs can execute the code in a distributed computing platform. When the application has finished, it restores the user code and saves the auto-generated code in a separated file.

For instance, Listing 4 shows the relevant parts of an Embarrassingly Parallel application with the main function annotated with the `@parallel` decorator that contains two nested loops.

```
1  # Main Function
2  from pycompss.api.parallel import parallel
3  @parallel()
4  def ep(mat, n_size, m_size, c1, c2):
5      for i in range(n_size):
6          for j in range(m_size):
7              mat[i][j] = compute(mat[i][j], c1, c2)
```

Listing 4: EP example: user code.

In addition, Listing 5 shows the parallelisation proposed by the AutoParallel module. On the one hand, the automatically generated source code contains the definition of a new task (*S1*) that includes the task decorator, annotations for its data dependencies (line 2), and the function code (line 4). Notice that the function code is automatically generated from the inner statement in the original loop (line 7 in Listing 4).

```
1  # [COMPSs Autoparallel] Begin Autogenerated code
2  @task(var2=IN, c1=IN, c2=IN, returns=1)
3  def S1(var2, c1, c2):
4      return compute(var2, c1, c2)
5
6  def ep(mat, n_size, m_size, c1, c2):
7      if m_size >= 1 and n_size >= 1:
8          lbp = 0
9          ubp = m_size - 1
10         for t1 in range(lbp, ubp + 1):
11             lbv = 0
12             ubv = n_size - 1
13             for t2 in range(lbv, ubv + 1):
14                 mat[t2][t1]=S1(mat[t2][t1],c1,c2)
15     compss_barrier()
16 # [COMPSs Autoparallel] End Autogenerated code
```

Listing 5: EP example: auto-generated code.

On the other hand, the generated source code contains the *ep* function with some modifications. First, AutoParallel introduces a new set of variables (e.g., *lbp*, *ubp*, *lbv*, *ubv*) to control the bounds of each loop. Also, the loop nest is modified following the Pluto output to call the automatically generated tasks (line 14) and exploit the inherent parallelism available in the original code. For instance, in the example, the loop bounds have been interchanged (*n_size* and *m_size*). Finally, AutoParallel includes a barrier (line 15) used as synchronisation point at the end of the function code.

4.2 Taskification of loop tiles

Many compute-intensive scientific applications are not designed as block computations, and thus, the tasks proposed by the AutoParallel module are single statements. Although this can be harmless in tiny parallel environments, it leads to poor performance when executed using large distributed environments since the tasks' granularity is not large enough to surpass the overhead of transferring the task definition, and the input and output data. To face this issue, we have extended the *Python to PyCOMPSs Translator* to automatically build data blocks from loop tiles and taskify them. As shown in Listing 6, the users can enable this behaviour by providing the `tile=True` option to the `@parallel` decorator.

```
1  from pycompss.api.parallel import parallel
2  @parallel(tile=True)
3  def ep(mat, n_size, m_size, c1, c2):
4      for i in range(n_size):
5          for j in range(m_size):
6              mat[i][j] = compute(mat[i][j], c1, c2)
```

Listing 6: EP example: user code with taskification of loop tiles.

Essentially, the taskification of loop tiles means letting Pluto process the parallel code by generating tiles, and extract the loop tiles into tasks. As explained in Section 3.2.1, the tiled loops generated by Pluto duplicate the depth of the original loop nest while decreasing the number of iterations per loop. Hence, extracting all the tiles and converting them into tasks maintains the original loop depth, decreases the number of iterations per loop, and increases the task granularity.

Since tasks may use N-dimensional arrays, the taskification of loop tiles also implies to create the necessary data blocks (chunks) for each parameter before the task callee. The automatically generated code builds the data blocks on the main code and passes them to the tasks using the PyCOMPSs Collection parameter annotation. This annotation ensures that all the internal objects of the chunks are registered so that all the parameter dependencies are respected. Following with the previous example, Listing 7 shows the automatically generated code with taskification of loop tiles.

Notice that the generated code with taskification of loop tiles is significantly more complex. In the example, the original loop nest has depth 2, the tile size for the `t1` loop is set to 2, and the tile size for the `t2` loop is set to 32. Also, readers may identify the `t3` loop as the tile of `t2` loop, and the `t4` loop as the tile of the `t1` loop (indexes have been swapped automatically due to data locality).

```
1  # [COMPSs Autoparallel] Begin Autogenerated code
2  @task(t2=IN, m_size=IN, t1=IN, n_size=IN, coef1=IN,
3    coef2=IN, mat={Type: COLLECTION_INOUT, Depth: 2})
4  def LT2(t2, m_size, t1, n_size, coef1, coef2, mat):
5    for t3 in range(32*t2, min(m_size, 32*t2 + 32)):
6      lbv = 2*t1
7      ubv = min(n_size - 1, 2*t1 + 1)
8      for t4 in range(lbv, ubv + 1):
9        mat[t4 - 2*t1][t3 - 32*t2] = S1_no_task(
10         mat[t4 - 2*t1][t3 - 32*t2], coef1, coef2)
11
12 def S1_no_task(var2, coef1, coef2):
13   return compute(var2, coef1, coef2)
14
15 def ep(mat, n_size, m_size, coef1, coef2):
16   if m_size >= 1 and n_size >= 1:
17     lbp = 0
18     ubp = int(math.floor(float(n_size - 1)/float(2)))
19     for t1 in range(lbp, ubp + 1):
20       lbp = 0
21       ubp = int(math.floor(float(m_size - 1)/float(32)))
22       for t2 in range(lbp, ubp + 1):
23         lbp = 32 * t2
24         ubp = min(m_size - 1, 32*t2 + 31)
25         # Chunk creation
26         LT2_aux_0 = [[mat[gv0][gv1] for gv1 in ...]
27           for gv0 in ...]
28         # Task call
29         LT2(t2, m_size, t1, n_size, coef1, coef2,
30             LT2_aux_0)
31   compss_barrier()
32 # [COMPSs Autoparallel] End Autogenerated code
```

Listing 7: EP example: auto-generated code with taskification of loop tiles.

Regarding the tasks, `LT2` contains a loop nest of depth 2 with 2 iterations for the `t4` loop, and 32 iterations for the `t3` loop. Furthermore, each N-dimensional array used as a parameter is annotated as a Collection with its direction (IN or INOUT) and depth (number of dimensions). In the example, `mat` is annotated as a `COLLECTION_INOUT` of depth 2. Moreover, inside the task code, the array accesses are modified accordingly to the received data chunks.

Regarding the main code, the original loops are modified considering the tiles' decoupling. In the example, the number of iterations of the `t1` loop is divided by 2, and the number of iterations of the `t2` loop is divided by 32. Furthermore, the data chunks are built before the task callee by only copying the original object references. In the example, we build the `LT2_aux_0` chunk from `mat` and update the callee parameters accordingly. Finally, similarly to the previous cases, the end of the main code also includes a barrier as a synchronisation point.

4.3 Python Extension for CLooG

As described in Section 3.2, Pluto operates internally with the OpenScop format. It relies on Clan to translate input code from C/C++ or Fortran to OpenScop, and on CLooG to translate output code from OpenScop to C/C++ or Fortran.

Since we are targeting Python code, a translation from Python to OpenScop is required before calling Pluto, and another translation from OpenScop to Python is required at the end. Regarding the input, we have developed the *Python To OpenScop Translator* component inside AutoParallel to manually translate the code because Clan is not adapted for supporting additional languages and Pluto accepts OpenScop codes as input. Regarding the output, we have extended CLooG so that the written code is directly in Python. Hence, we have extended the language options and modified the Pretty Printer in order to translate every OpenScop statement into its equivalent Python format. Since Python does not have any standard regarding parallel annotations, the parallel loops are annotated with comments in OpenMP syntax.

5 Programmability Evaluation

Considering an idealistic environment, the developer's productivity can be expressed as the relation between the effort to write the code of the application and the performance obtained by such code. We are aware that many other factors such as the physical working environment, adequate development frameworks and tools, meeting times, code reviews, burndowns, etc. can affect the developer's productivity, but we only consider elements directly related with the code. Furthermore, these terms can be considered constants or eventualities when comparing the productivity difference for the same developer when using (or not) the AutoParallel module.

In this section, we demonstrate that our approach improves the developer's productivity significantly by easing the coding of the application. On the other hand, next sections (Section 6 and 7) focus on evaluating and comparing the performance of the automatically generated codes. Hence, the application presented in this section highlights the benefits at the programming model level of using AutoParallel without focusing on performance.

5.1 Centre of Mass

The following application calculates the centre of mass of a given system. The *system* itself is composed of *objects* that are composed of *parts* in a certain *position* of the space. Also, each part has a predefined *mass*. Equation 1 describes how to compute the centre of mass of the system (CM) by first calculating the centre of mass and the aggregated mass of each object and, next, computing the centre of mass of the whole system.

$$
\begin{aligned}
cm_{obj} &= \frac{\sum_{j=0}^{num\_parts} mass_{obj,j} \cdot position_j}{\sum_{j=0}^{num\_parts} mass_j} \\
mass_{obj} &= \sum_{j=0}^{num\_parts} mass_j \\
CM &= \frac{\sum_{i=0}^{num\_objs} mass_i \cdot cm_i}{\sum_{i=0}^{num\_objs} mass_i}
\end{aligned}
\tag{1}
$$

Listing 8 provides the sequential code to calculate the centre of mass following Equation 1. The first loop nest (lines 11 to 26 in the figure) calculates the numerator and the denominator of cm_{obj} so that the second loop nest (lines 32 to 39) can compute the centre of mass of each object in the system. Then, the third loop nest (lines 42 to 45) computes the centre of mass of the whole system.

Given the sequential code, the users can add the `@parallel` decorator to automatically taskify and run the application in a distributed environment. Listing 9 shows that the AutoParallel module can be enabled by just adding 2 lines (the import and the decorator) on top of the method declaration.

```python
def calculate_cm(num_objs, num_parts, num_dims,
                 objs, masses):
  import numpy as np

  # Initialize object results
  objs_cms = [[np.float(0) for _ in range(num_dims)]
              for _ in range(num_objs)]
  objs_mass = [np.float(0) for _ in range(num_objs)]

  # Calculate CM and mass of every object
  for obj_i in range(num_objs):
    # Calculate object mass and cm position
    for part_i in range(num_parts):
      # Update total object mass
      objs_mass[obj_i]
        += masses[objs[obj_i][part_i][0]]
      # Update total object mass position
      for dim in range(num_dims):
        objs_cms[obj_i][dim]
          += masses[objs[obj_i][part_i][0]]
             * objs[obj_i][part_i][1][dim]
    # Store final object CM and mass
    for dim in range(num_dims):
      objs_cms[obj_i][dim] /= objs_mass[obj_i]
        if objs_mass[obj_i] != np.float(0)
        else np.float(0)

  # Initialize system results
  system_mass = np.float(0)
  system_cm = [np.float(0) for _ in range(num_dims)]

  # Calculate system CM for every object
  for obj_i in range(num_objs):
    # Update total mass
    system_mass += objs_mass[obj_i]
    # Update system mass position
    for dim in range(num_dims):
      system_cm[dim] += objs_mass[obj_i]
                        * objs_cms[obj_i][dim]

  # Calculate system CM
  for dim in range(num_dims):
    system_cm[dim] /= system_mass
      if system_mass != np.float(0)
      else np.float(0)

  return system_cm
```

Listing 8: Centre of mass: sequential code.

```python
from pycompss.api.parallel import parallel

@parallel()
def calculate_cm(num_objs, num_parts, num_dims,
                 objs, masses):
  # Exactly the same code than the original
  ...

  return system_cm
```

Listing 9: Centre of mass: AutoParallel annotations.

As we stated in Section 2, other programming models such as NumPy offer vectorised data structures and operators in a transparent way to prevent the users from defining loops to handle NumPy values directly. Although the AutoParallel module can automatically parallelise and distributedly execute loops containing NumPy vectorised operations (taskifying them as any other single statement), the purpose of this section is to compare the AutoParallel module against pure sequential Python code without considering additional libraries that can benefit from intra-node parallelism.

Next, Table 1 compares the user code and the automatically generated code in terms of annotations and loop configuration.

Version	Code Analysis			Loops Analysis		
	Annotations		API Calls	Main	Total	Max Depth
	Method	Param.				
autoparallel (user code)	1	0	0	3	7	3
autoparallel (generated)	6	15	3	14	29	3

Table 1. Centre of mass: code and loop analysis.

On the one hand, although PyCOMPSs' annotations are quite simple compared to other programming models (such as MPI), the automatically generated code contains 6 different task definitions with 15 parameter annotations. More in-depth, Listing 10 details each task definition where *S1* corresponds to the statement in line 15 of the sequential code (Listing 8), *S2* to line 19, *S3* to line 24, *S4* to line 35, *S5* to line 38, and *S6* to line 43. Furthermore, notice that AutoParallel creates a new task per statement in the original loop nest, even if the internal operation is the same at the end. Hence, a user manually parallelising and taskifying the previous sequential code will obtain a similar solution but using 3 tasks instead of 6 since *S1* and *S4*, *S2* and *S5*, and *S3* and *S6* can be merged.

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

@task(var2=IN, var1=INOUT)
def S1(var2, var1):
  var1 += var2

@task(var2=IN, var3=IN, var1=INOUT)
def S2(var2, var3, var1):
  var1 += var2 * var3

@task(var2=IN, var3=IN, var1=INOUT)
def S3(var2, var3, var1):
  var1 /= var3 if var2 != np.float(0)
               else np.float(0)

@task(var1=IN, system_mass=INOUT)
def S4(var1, system_mass):
  system_mass += var1

@task(var2=IN, var3=IN, var1=INOUT)
def S5(var2, var3, var1):
  var1 += var2 * var3

@task(system_mass=IN, var1=INOUT)
def S6(system_mass, var1):
  var1 /= system_mass if system_mass != np.float(0)
                      else np.float(0)
```

Listing 10: Centre of mass: Automatically generated tasks.

On the other hand, AutoParallel re-writes the loop nests in order to exploit data locality and perform some optimisations depending on the input values. Hence, the 3 original loop nests have been split into 14 loop nests containing a total of 29 for loops. However, notice that the maximum depth (3) is preserved to ensure that the complexity remains the same. More in-depth, Listing 11 shows the automatically generated loop nest, including the task and data synchronisation calls. Due to space constraints, we only include the generated code for the second and third loop nests (lines 32 to 39 and 42 to 45 in the sequential code on Listing 8) since the first one is more complex and has been divided into 10 loop nests.

Regarding the second loop nest, *S5* computes the nominator of the system's centre of mass and *S4* the

denominator (total mass of the system). Also, AutoParallel has split the main loop into 3 loops (lines 23, 36, and 40 in Listing 11) considering different input values and iterating in a different way over the loop space to exploit data locality.

Regarding the third loop nest, although AutoParallel has automatically added the *lbp* and *ubp* variables to control the loop bounds, the loop structure is kept the same. Furthermore, the original statement is substituted by a task call to *S6*.

After the code corresponding to each original loop nest, AutoParallel automatically adds a `compss_barrier()` to avoid possible data collisions (lines 46 and 54 in Listing 11). Also, there is a synchronisation of the `system_cm` variable (line 56) before the return of the function so that PyCOMPSs synchronises and transfers its final value.

```
1  from pycompss.api.api import compss_wait_on
2
3  def calculate_cm(num_objs, num_parts, num_dims,
4                   objs, masses):
5    import numpy as np
6
7    # Initialize object results
8    objs_cms = [[np.float(0) for _ in range(num_dims)]
9                for _ in range(num_objs)]
10   objs_mass = [np.float(0) for _ in range(num_objs)]
11
12   # Calculate CM and mass of every object
13   ...
14
15   # Calculate system CM for every object
16   system_mass = np.float(0)
17   system_cm = [np.float(0) for _ in range(num_dims)]
18
19   if num_objects >= 1:
20     if num_objects >= 2:
21       lbp = 0
22       ubp = min(num_dims - 1, num_objects - 1)
23       for t1 in range(lbp, ubp + 1):
24         S4(objs_mass[t1], system_mass)
25         S5(objs_mass[0], objs_cms[0][t1], system_cm[t1])
26         lbp = 1
27         ubp = num_objects - 1
28         for t2 in range(1, num_objects - 1 + 1):
29           S5(objs_mass[t2], objs_cms[t2][t1],
30              system_cm[t1])
31     if num_dims >= 1 and num_objects == 1:
32       S4(objs_mass[0], system_mass)
33       S5(objs_mass[0], objs_cms[0][0], system_cm[0])
34     lbp = max(0, num_dims)
35     ubp = num_objects - 1
36     for t1 in range(lbp, ubp + 1):
37       S4(objs_mass[t1], system_mass)
38     lbp = num_objects
39     ubp = num_dims - 1
40     for t1 in range(lbp, ubp + 1):
41       lbp = 0
42       ubp = num_objects - 1
43       for t2 in range(0, num_objects - 1 + 1):
44         S5(objs_mass[t2], objs_cms[t2][t1],
45            system_cm[t1])
46   compss_barrier()
47
48   # Calculate system CM
49   if num_dims >= 1:
50     lbp = 0
51     ubp = num_dims - 1
52     for t1 in range(lbp, ubp + 1):
53       S6(system_mass, system_cm[t1])
54   compss_barrier()
55
56   system_cm = compss_wait_on(system_cm)
57
58   return system_cm
```

Listing 11: Centre of mass: Automatically generated loop nest.

To conclude, AutoParallel is capable of automatically taskifying a sequential code and re-order the loop nests to exploit data locality by just adding one single annotation. For the centre of mass application, AutoParallel frees the users from defining 6 tasks, annotating 15 parameters, building each task call and re-ordering the *objects*, *parts*, and *dimensions* loops to exploit data locality. Also, notice that PyCOMPSs already provides a simple programming model compared to many other frameworks such as MPI. In those cases, AutoParallel frees the users from explicitly defining the code for each process, handling data transfers, and synchronising the different processes.

6 Performance Evaluation

This section demonstrates that our approach eases the coding of the application using one single Python decorator on top of sequential code, while obtaining similar performances than manually parallelised codes. Next subsections evaluate the code complexity and the performance for established algorithms when using AutoParallel or PyCOMPSs. We evaluate the Cholesky, LU, and QR decompositions in order to demonstrate that AutoParallel is capable of automatically generating code as efficient as solutions that have been a reference in state of the art for more than ten years, but requiring much less effort to write them.

It is worth highlighting that AutoParallel is an additional module to generate PyCOMPSs annotated applications automatically but, at execution time, the generated code acts as a regular PyCOMPSs application. Hence, the execution performance is strongly related to the PyCOMPSs performance. This section reports the performance evaluation of several applications in comparison with their implementations using only PyCOMPSs to evaluate only the overhead introduced by AutoParallel. Therefore, the performance evaluation of PyCOMPSs and its Runtime is beyond the scope of this paper. For further details, in our previous work Amela and et al. (2017) and Amela and et al. (2018), we analysed in-depth the performance obtained when executing linear algebra applications when combining PyCOMPSs for inter-node parallelism and NumPy for intra-node parallelism. Also, in Conejero and et al. (2018), we compared the PyCOMPSs Runtime against Apache Spark.

6.1 Computing Infrastructure

Results presented in this section have been obtained using the MareNostrum IV Supercomputer located at the Barcelona Supercomputing Center (BSC).

We have used PyCOMPSs version 2.6 (available at Barcelona Supercomputing Center (BSC) (2019a)), Pluto version 0.11.4, CLooG version 0.19.0, and AutoParallel version 1.0 (available at Ramon-Cortes (2019b)). We have also used Intel®Python 2.7.13, Intel®MKL 2017, Java OpenJDK 8 131, GCC 7.2.0, and Boost 1.64.0.

All the benchmark codes used for this experimentation are also available at Ramon-Cortes (2019a).

MareNostrum IV The MareNostrum IV begun operating at the end of June 2017. Its current peak performance is 11.15 Petaflops, ten times more than its previous version, MareNostrum III. The supercomputer is composed by 3456

nodes, each of them with two Intel® Xeon Platinum 8160 (24 cores at 2.1 GHz each). It has 384.75 TB of main memory, 100Gb Intel®Omni-Path Full-Fat Tree Interconnection, and 14 PB of disk storage Barcelona Supercomputing Center (BSC) (2019c).

6.2 General description of the applications

In general terms, the matrices are chunked in smaller square matrices (known as *blocks*) to distribute the data easily among the available resources so that the square blocks are the minimum entity to work with Gunnels and et al. (2001). Furthermore, the initialisation is performed in a distributed way, defining tasks to initialise the matrix blocks. These tasks do not take into account the nature of the algorithm, and they are scheduled in a round robin manner. For all the evaluation applications, the execution time measures the application's computations and the data transfers required during the execution by the framework, but does not include the initial transfers of the input data.

Given a fixed matrix size, increasing the number of blocks increases the maximum parallelism of the application since blocks are the tasks' minimum work entities. On the other hand, increasing the block size increases the tasks' computational load, which, at some point, will surpass the serialisation and transfer overheads. Hence, the number of blocks and the block size for each application are a trade-off to fill all the available cores while maintaining acceptable performance.

For all the evaluated applications, we compare the code written by a PyCOMPSs expert user (*userparallel* version) against the sequential code with the `@parallel` annotation (*autoparallel* user code) and the automatically generated code by the AutoParallel module (*autoparallel* generated). Furthermore, we provide a figure showing the execution results for each application. The figure contains two plots where the horizontal axis shows the number of worker nodes (with 48 cores each) used for each execution, the blue colour is the *userparallel* version, and the green colour is the *autoparallel*. The top plot represents the mean, maximum, and minimum execution times over 10 runs and the bottom plot represents the speed-up of each version with respect to the *userparallel* version running with a single worker (48 cores).

6.3 Cholesky

The Cholesky factorisation can be applied to Hermitian positive-defined matrices. This decomposition is a particular case of the LU factorisation, obtaining two matrices of the form $U = L^t$. Our version of this application applies the right-looking algorithm Bientinesi et al. (2008) 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.

Table 2 analyses the *userparallel*, the *autoparallel*'s original user code, and the *autoparallel*'s automatically generated code in terms of code and loop configuration. While the *userparallel* version requires the definition of three tasks (`potrf`, `solve_triangular`, and `gemm`) using 14 parameter annotations, the *autoparallel*'s original code only requires a single `@parallel` decorator. On the other hand, the *autoparallel*'s automatically generated code includes four tasks; 3 equivalent to the *userparallel* tasks and an additional one to generate blocks initialised to zero.

Version	Code Analysis			Loops Analysis		
	Annotations		API	Main	Total	Max
	Method	Param.	Calls			Depth
userparallel	3	14	0	1	4	3
autoparallel (user code)	1	0	0	1	4	3
autoparallel (generated)	4	11	1	3	9	3

Table 2. Cholesky: code and loop analysis.

Regarding the loop configuration, the *userparallel* and the *autoparallel*'s original code have the same structure. However, the *autoparallel*'s automatically generated code has divided the original loop into three main loops maintaining the maximum loop depth (three).

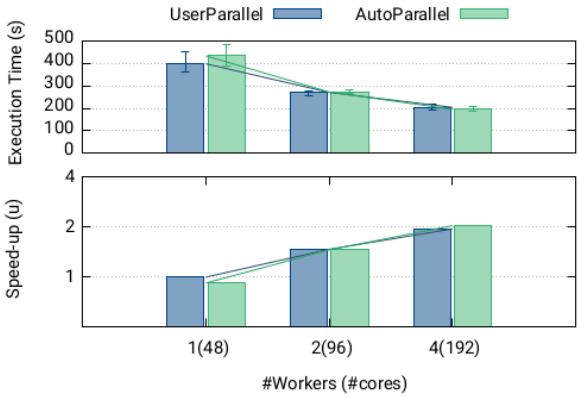

Figure 5. Cholesky: Execution times and speed-up with respect to the *userparallel* version using a single worker (48 cores).

Figure 5 shows the execution results of the Cholesky decomposition over a dense matrix of $65,536 \times 65,536$ elements decomposed in 32×32 blocks with $2,048 \times 2,048$ elements each. As stated in the general description of this section, the top plot represents the execution time while the bottom plot represents the speed-up with respect to the *userparallel* version running with a single worker (48 cores). Also, we have chosen 32 blocks because it is the minimum amount providing enough parallelism for 192 cores, and bigger block sizes (e.g., $4,096 \times 4,096$) were impossible due to memory constraints. The speed-up of both versions is limited by the block-size due to the small task granularity, reaching 2 when using 4 workers. Although the *userparallel* version spawns 6,512 tasks and the *autoparallel* version spawns 7,008 tasks, the execution times and the overall performance of both versions are almost the same (the difference is less than 5%). This is due to the fact that the *autoparallel* version spawns an extra task per iteration to initialise blocks to zero on the matrix's lower triangle that has no impact on the overall computation time.

6.4 LU

For the LU decomposition, an approach without pivoting Golub and Van Loan (1996) has been the starting

point. However, since this approach might be unstable in general Demmel and Higham (1992), some modifications have been included to increase the stability of the algorithm while keeping the block division and avoiding bringing an entire column into a single node.

Version	Code Analysis			Loops Analysis		
	Annotations		API	Main	Total	Max
	Method	Param.	Calls			Depth
userparallel	4	13	0	2	6	3
autoparallel (user code)	1	0	0	2	6	3
autoparallel (generated)	12	33	3	4	9	3

Table 3. LU: code and loop analysis.

Table 3 analyses the *userparallel*, the *autoparallel*'s original user code, and the *autoparallel*'s automatically generated code in terms of code and loop configuration. Regarding the code, the *userparallel* version requires the definition of 4 tasks (namely `multiply`, `invert_triangular`, `dgemm`, and `custom_lu`) along with 13 annotated parameters. In contrast, the *autoparallel*'s original user code only requires the `@parallel` annotation. Also, the *autoparallel*'s automatically generated code generates 12 different task types (along with 33 annotated parameters) because it generates one task type per statement in the original loop, even if the statement contains the same task call. For instance, the original LU contains four calls to the `invert_triangular` function that are detected as different statements and converted to different task types.

Regarding the loop configuration, both the *userparallel* and the*autoparallel*'s original user code have the same structure: 2 loop nests of depth 3. However, the *autoparallel*'s automatically generated code splits them into 4 main loops of the same depth because it has different optimisation codes for different variable values.

Figure 7 shows the execution results of the LU decomposition with a $49,152 \times 49,152$ dense matrix of 24×24 blocks with $2,048 \times 2,048$ elements each. As stated in the general description of this section, the top plot represents the execution time while the bottom plot represents the speed-up with respect to the *userparallel* version running with a single worker (48 cores). As in the previous example, the overall performance is limited by the block size. This time the *userparallel* version slightly outperforms the *autoparallel* version; achieving, respectively, a 2.45 and 2.13 speed-up with 4 workers (192 cores).

Regarding the number of tasks, the *userparallel* version spawns 14,676 tasks while the *autoparallel* version spawns

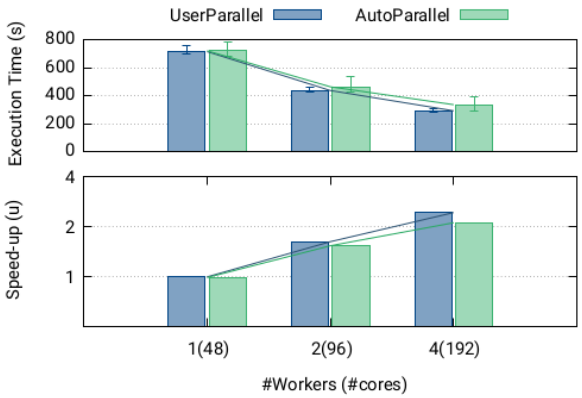

Figure 7. LU: Execution times and speed-up with respect to the *userparallel* version using a single worker (48 cores).

15,227 tasks. This difference is due to the fact that the *autoparallel* version initialises distributedly an intermediate zero matrix, while the *userparallel* initialises it in the master memory.

Figure 6 shows a detailed Paraver trace of both versions running with 4 workers (192 cores). The *autoparallel* version (right) is more coloured because it has more tasks, although, as previously explained, they execute the same function in the end. Notice that the performance degradation of the *autoparallel* version is due to the fact that the maximum parallelism is lost before the end of the execution. On the contrary, the *userparallel* version maintains the maximum parallelism until the end of the execution.

6.5 QR

Unlike traditional QR algorithms that use the Householder transformation, our implementation uses a method based on Givens rotations Quintana-Orti and et al. (2008). This way, data can be accessed by blocks instead of columns.

Version	Code Analysis			Loops Analysis		
	Annotations		API	Main	Total	Max
	Method	Param.	Calls			Depth
userparallel	4	19	0	1	6	3
autoparallel (user code)	1	0	0	1	6	3
autoparallel (generated)	20	60	1	2	7	3

Table 4. QR: code and loop analysis.

Table 4 analyses the *userparallel*, the *autoparallel*'s original user code, and the *autoparallel*'s automatically generated code in terms of code and loop configuration. The QR decomposition represents one of the most complex use

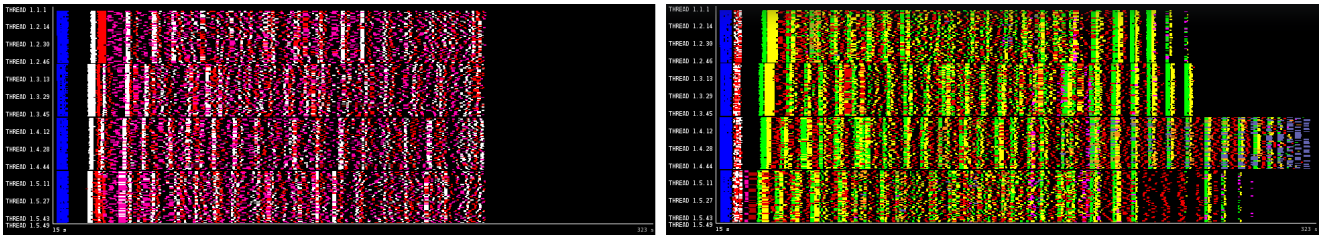

Figure 6. LU: Paraver trace. At left, *userparallel* and, at right, the *autoparallel* version.

cases in terms of data dependencies; thus, having more tasks and parameter annotations than the previous applications. While the *userparallel* requires 4 tasks (namely `qr`, `dot`, `little_qr`, and `multiply_single_block`) along with 19 parameter annotations, the *autoparallel*'s original user code only requires, as always, one single `@parallel` annotation. In contrast, the *autoparallel*'s automatically generated code defines 20 tasks along with 60 parameter annotations. As in the LU decomposition, many of these tasks are generated from different statements that perform the same task call at the end.

Regarding the loop configuration, both the *userparallel* and the*autoparallel*'s original user code have the same structure: 1 main loop nest with a total of 6 loops and a maximum depth of 3. However, the *autoparallel*'s automatically generated code splits the main loop in two in order to increase the data locality. Notice that no additional complexity is added to the algorithm since the maximum depth remains 3.

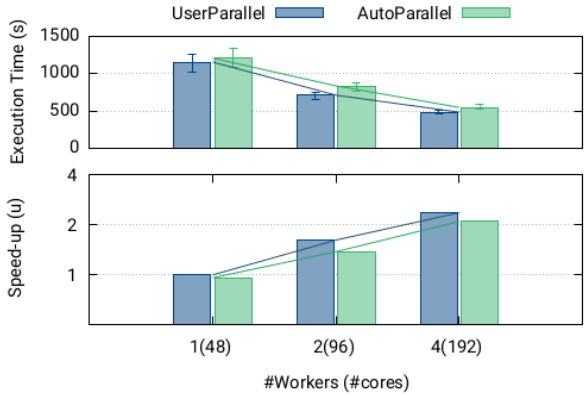

Figure 8. QR: Execution times and speed-up with respect to the *userparallel* version using a single worker (48 cores).

Figure 8 shows the execution results of the QR decomposition with a $32,768 \times 32,768$ matrix of 16×16 blocks with $2,048 \times 2,048$ elements each. As stated in the general description of this section, the top plot represents the execution time while the bottom plot represents the speed-up with respect to the *userparallel* version running with a single worker (48 cores). The *autoparallel* version spawns 26,304 tasks, and the *userparallel* version spawns 19,984 tasks. As in the previous examples, the overall performance is limited by the block size. However, the *userparallel* version slightly outperforms the *autoparallel* version; achieving a 2.37 speed-up with 4 workers instead of 2.10. The difference is mainly because the *autoparallel* version spawns four copy tasks per iteration (`copy_reference`), while the *userparallel* version executes this code in the master side copying only the reference of a future object.

7 Evaluation of the automatic data blocking

This section evaluates the capability of automatically generating data blocks (chunks) from pure sequential code and executing them in a distributed infrastructure. As discussed next, this approach provides several advantages in terms of code re-organisation and data blocking; increasing the tasks' granularity and, thus, the performance of fine-grain applications.

7.1 GEMM

We have implemented a Python version of the General Matrix-Matrix product (GEMM) from the Polyhedral Benchmark suite The Ohio State University, Department of Computer Science and Engineering (2015). The implementation considers general rectangular matrices with float complex elements and performs $C = \alpha \cdot A \cdot B + \beta \cdot C$. In general terms, the arrays and matrices are implemented as plain NumPy arrays or matrices. This means that there are no *blocks*, and thus, the minimum work entity is a single element (a float). As in the previous set of experiments, the initialisation is performed in a distributed way; defining tasks to initialise the matrix elements. Also, the execution time measures the application's computations and the data transfers required during the execution by the framework, but does not include the initial transfers of the input data.

Version	Code Analysis			Loops Analysis		
	Annotations		API Calls	Main	Total	Max Depth
	Method	Param.				
userparallel	2	8	0	1	4	3
autoparallel (user code)	1	0	0	1	4	3
autoparallel FG (generated)	2	8	1	2	5	3
autoparallel LT (generated)	4	21	1	2	5	3

Table 5. GEMM: code and loop analysis.

Table 5 analyses the *userparallel*, the *autoparallel*'s original user code, the *autoparallel*'s automatically generated code using fine-grain (*autoparallel FG*), and the *autoparallel*'s automatically generated code with taskification of loop tiles (*autoparallel LT*) in terms of code and loop configuration. As expected, the *autoparallel*'s original user code only requires the `@parallel` annotation. Although the *autoparallel FG* works with single elements and *userparallel* with blocks, both versions include 2 tasks (namely `scale`, and `multiply`) with 8 parameter annotations. On the contrary, the *autoparallel LT* version defines 4 tasks (the two original ones and their two loop-tasked versions) with 21 parameter annotations. The original tasks are kept because, in configurations that do not use Pluto's tiles, it is possible to find function calls that cannot be loop-taskified. However, in this case, only the loop-tasked versions are called during the execution.

Regarding the loop structure, both the *userparallel* and the *autoparallel*'s original user code have 1 main loop of maximum depth 3. Also, the two automatically generated codes are capable of splitting the main loop into two loops for better parallelism: one for the scaling operations and the other for the multiplications. However, the *autoparallel LT* code is significantly more complex (in terms of lines of code, cyclomatic complexity, and n-path) due to the tiling and chunk creation.

Figure 9 shows the execution results of the GEMM application with one single worker (48 cores) and with matrices of 8, 16, 32, and 64 elements. To have equivalent

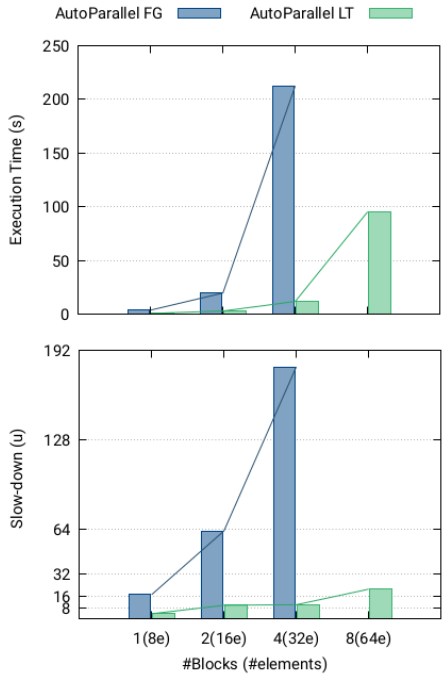

Figure 9. GEMM: Execution time and slow-down with respect to the blocked *userparallel* version using a single worker (48 cores) and the same matrix size.

executions, the tile sizes are set to 8 for the *autoparallel LT*, and the block size is set to 8 for the *userparallel*. The top plot shows the execution time of the *autoparallel FG* (blue) and the *autoparallel LT* (green). The bottom plot shows the slow-down of both versions with respect to the blocked *userparallel* version using a single worker (48 cores) and the same matrix size.

The automatic parallelisation without taskification of loop tiles (*autoparallel FG*) behaves 17.30 and 179.94 times slower than the blocked version (*userparallel B*) using 8 and 32 elements, respectively. In contrast, the *autoparallel LT* behaves 10.15 times slower than the *userparallel B* when using 32 elements; which improves by 17.73 the performance of the fine-grain version. This experiment highlights the importance of blocking fine-grain applications since defining single elements as the minimum task entity leads to tasks with too little computation that cause a massive overhead of task management, object serialisation and, data transfer inside PyCOMPSs.

Due to this same reason and as shown in the previous Section 6, the appropriate block sizes to obtain reasonable performances should be between $2,048 \times 2,048$ and $8,192 \times 8,192$ elements per block. Although AutoParallel can generate codes using bigger tile sizes, PyCOMPSs suffers from serialisation issues since each element inside the collection is treated as a separated task parameter. In contrast, the hand-made blocked version (*userparallel*) serialises all the elements of the block into a single object parameter and thus, can run with bigger block sizes. We are confident that PyCOMPSs could lower the serialisation overhead by serialising each element of the collection in parallel or serialising the whole collection into a single object.

Nonetheless, we believe that the automatic taskification of loop tiles is a good baseline to obtain blocked algorithms since the only difference between the automatically generated code with taskification of loop tiles and the hand-made blocked version is the treatment of data chunks. More specifically, the *userparallel* version uses single objects per chunk instead of collections of objects. Notice that AutoParallel cannot systematically annotate data chunks as objects since this is only possible when the data chunks are disjoint because, otherwise, the dependencies of each element inside the blocks need to be treated separately. However, advanced users can analyse the automatically generated data chunks, determine if they are disjoint, and use the automatically generated code as a baseline to change the annotations of the data chunks from `COLLECTION` to `OBJECT`. In this last scenario, advanced users only require to modify annotation of the data chunks; keeping the main code of the algorithm and the code of the tasks.

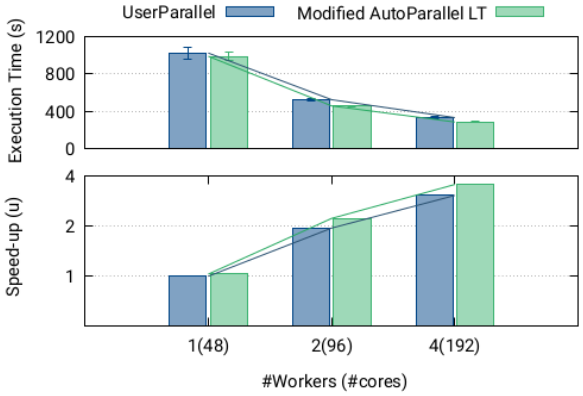

Figure 10. GEMM with object annotations: Execution time and speed-up with respect to the blocked *userparallel* version using a single worker (48 cores).

For instance, Figure 10 compares the execution results of the *userparallel* blocked version and the *autoparallel LT* version changing the collection annotations per object annotations when running the GEMM application over a dense matrix of $65,536 \times 65,536$ elements decomposed in 32×32 blocks with $2,048 \times 2,048$ elements each. The top plot represents the execution time while the bottom plot represents the speed-up with respect to the *userparallel* blocked version running with a single worker (48 cores). Also, we have chosen 32 blocks because it is the minimum amount providing enough parallelism for 192 cores, and bigger block sizes (e.g., $4,096 \times 4,096$) were impossible due to memory constraints.

We must highlight that the modified *autoparallel LT* version outperforms the *userparallel* version because it is capable of better exploiting the parallelisation of the scaling operation. With little modifications regarding the parameter annotations, advanced users can obtain better blocked algorithms than manually parallelised codes. Hence, we believe that the automatic taskification of loop tiles is a good baseline to design complex blocked algorithms.

8 Conclusions and Future Work

This paper has presented and evaluated AutoParallel, a Python module to automatically parallelise affine loop nests and execute them on distributed infrastructures. Built on

top of PyCOMPSs and Pluto, it is based on sequential programming so that anyone can scale up an application to hundreds of cores. Instead of manually taskifying a sequential python code, the users only need to add a `@parallel` annotation to the methods containing affine loop nests.

The evaluation shows that the codes automatically generated by the AutoParallel module for the Cholesky, LU, and QR applications can achieve similar performance than manually parallelised versions without requiring any effort from the programmer. Thus, AutoParallel goes one step further in easing the development of distributed applications.

Furthermore, the taskification of loop tiles can be enabled by providing a single decorator argument (`tile=True`) to automatically build data blocks from loop tiles and increase the tasks' granularity. Although the overhead with respect to the hand-made blocked version is still far from acceptable, the taskification of loop tiles provides an automatic way to build data blocks from any application; freeing the users from dealing directly with the complexity of block algorithms and allowing them to stick to basic sequential programming. Also, for advanced users, the generated code can be used as a baseline to modify the annotations of the data chunks and obtain the same performance than the hand-made blocked version when the data chunks are disjoint.

As future work, we believe that the taskification of loop tiles is a good approach for fine-grain applications provided that the serialisation performance is improved. For instance, PyCOMPSs could lower the serialisation overhead by serialising each element of the collection in parallel or by serialising the whole collection into a single object.

Finally, AutoParallel could be integrated with different tools similar to Pluto to support a broader scope of loop nests. For instance, Apollo Sukumaran-Rajam and Clauss (2015); Martinez Caamaño and et al. (2017) provides automatic, dynamic and speculative parallelisation and optimisation of programs' loop nests of any kind (for, while or do-while loops). However, its integration would require PyCOMPSs to be extended with some speculative mechanisms.

Acknowledgements

Thanks to Cédric Bastoul for his support with CLooG.

Funding

This work has been supported by the Spanish Government through contracts SEV2015-0493 and TIN2015-65316-P, and by Generalitat de Catalunya through contract 2014-SGR-1051. Cristian Ramon-Cortes predoctoral contract is financed by the Ministry of Economy and Competitiveness under the contract BES-2016-076791.

References

Amela R and et al (2017) Enabling Python to Execute Efficiently in Heterogeneous Distributed Infrastructures with PyCOMPSs. In: *Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing*. ACM, pp. 1:1–1:10. DOI:10.1145/3149869.3149870.

Amela R and et al (2018) Executing linear algebra kernels in heterogeneous distributed infrastructures with PyCOMPSs. *Oil —& Gas Science and Technology - Revue d'IFP Energies Nouvelles (OGST)* DOI:10.2516/ogst/2018047.

Anaconda (2020) Numba: A High Performance Python Compiler. http://numba.pydata.org. Cited 8 April 2020.

Anderson E and et al (1999) *LAPACK Users' guide*. SIAM.

Apache Software Fundation (2019) PySpark. https://spark.apache.org/docs/latest/api/python/index.html. Cited 8 October 2019.

Badia RM and et al (2015) COMP superscalar, an interoperable programming framework. *SoftwareX* 3: 32–36. URL https://doi.org/10.1016/j.softx.2015.10.004.

Barcelona Supercomputing Center (BSC) (2019a) COMPSs GitHub. https://github.com/bsc-wdc/compss. Cited 8 October 2019.

Barcelona Supercomputing Center (BSC) (2019b) Extrae Tool. https://tools.bsc.es/extrae. Cited 9 October 2019.

Barcelona Supercomputing Center (BSC) (2019c) MareNostrum IV Technical Information. https://www.bsc.es/marenostrum/marenostrum/technical-information. Cited 8 October 2019.

Barcelona Supercomputing Center (BSC) (2019d) Paraver Tool. https://tools.bsc.es/paraver. Cited 8 October 2019.

Barcelona Supercomputing Center (BSC) (2020) PyCOMPSs User Manual. https://compss-doc.readthedocs.io/en/2.6/Sections/02_User_Manual_App_Development.html#python-binding. Cited 8 April 2020.

Bastoul C (2004) Code Generation in the Polyhedral Model Is Easier Than You Think. In: *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*. IEEE Computer Society, pp. 7–16. URL https://doi.org/10.1109/PACT.2004.11.

Bastoul C (2011) OpenScop: A Specification and a Library for Data Exchange in Polyhedral Compilation Tools. Technical report, Paris-Sud University, France. URL http://icps.u-strasbg.fr/people/bastoul/public_html/development/openscop/docs/openscop.html.

Bastoul C and et al (2003) Putting Polyhedral Loop Transformations to Work. In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer, Springer Berlin Heidelberg, pp. 209–225. URL https://doi.org/10.1007/978-3-540-24644-2_14.

Bientinesi P, Gunter B and van de Geijn RA (2008) Families of Algorithms Related to the Inversion of a Symmetric Positive Definite Matrix. *ACM Trans. Math. Softw.* 35(1): 3:1–3:22. DOI:10.1145/1377603.1377606.

Bondhugula U (2017) Pluto. http://pluto-compiler.sourceforge.net. Cited 8 October 2019.

Bondhugula U and et al (2008a) A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *SIGPLAN Not.* 43(6): 101–113. URL http://doi.org/10.1145/1379022.1375595.

Bondhugula U and et al (2008b) Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In: *International Conference on Compiler Construction*. Springer Berlin Heidelberg, pp. 132–146. DOI:10.1007/978-3-540-78791-4_9.

Cass S (2019) The Top Programming Languages 2019: Python remains the big kahuna, but specialist languages hold their own. URL https://

spectrum.ieee.org/computing/software/the-top-programming-languages-2019. Cited 19 December 2019.

Cohen A and et al (2005) Facilitating the Search for Compositions of Program Transformations. In: *Proceedings of the 19th Annual International Conference on Supercomputing*. ACM, pp. 151–160. URL https://doi.org/10.1145/1088149.1088169.

Conejero J and et al (2018) Task-based programming in COMPSs to converge from HPC to big data. *The International Journal of High Performance Computing Applications* 32(1): 45–60. DOI:10.1177/1094342017701278.

Cooke DM, Alted F and et al (2020) NumExpr: Fast numerical expression evaluator for NumPy. https://github.com/pydata/numexpr. Cited 8 April 2020.

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

Dalcín L, Paz R and Storti M (2005) MPI for Python. *Journal of Parallel and Distributed Computing* DOI:https://doi.org/10.1016/j.jpdc.2005.03.010.

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

Demmel JW and Higham NJ (1992) Stability of Block Algorithms with Fast Level-3 BLAS. *ACM Trans. Math. Softw.* 18(3): 274–291. DOI:10.1145/131766.131769.

Golub GH and Van Loan CF (1996) *Matrix Computations (3rd Ed.)*. Baltimore, MD, USA: Johns Hopkins University Press. ISBN 0-8018-5414-8.

Google (2019) The Go Programming Language. https://golang.org/. Cited 8 October 2019.

Gunnels JA and et al (2001) FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Softw.* 27(4): 422–455. DOI:10.1145/504210.504213.

Intel (2019) Threading Building Blocks (Intel®TBB). https://software.intel.com/en-us/tbb. Cited 8 October 2019.

Jones E, Oliphant T and Peterson P (2001–) SciPy: Open source scientific tools for Python. URL http://www.scipy.org/.

Lam SK, Pitrou A and Seibert S (2015) Numba: A llvm-based python jit compiler. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. pp. 1–6.

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

Lordan F and et al (2014) ServiceSs: an interoperable programming framework for the Cloud. *Journal of Grid Computing* 12(1): 67–91. URL https://digital.csic.es/handle/10261/132141.

Martinez Caamaño JM and et al (2017) Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones. *Concurrency and Computation: Practice and Experience* 29(15): e4192. DOI:10.1002/cpe.4192.

McKinney W (2011) Pandas: a Foundational Python Library for Data Analysis and Statistics. *Python for High Performance and Scientific Computing* : 1–9.

Müller SC and et al (2014) Pydron: Semi-automatic parallelization for multi-core and the cloud. In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. pp. 645–659.

Python Software Fundation (2019) Parallel Processing and Multiprocessing in Python. https://wiki.python.org/moin/ParallelProcessing. Cited 8 October 2019.

Quintana-Orti G and et al (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. DOI:10.1109/PDP.2008.37.

Ramon-Cortes C (2019a) Experimentation GitHub. https://github.com/cristianrcv/pycompss-autoparallel/tree/master/examples. Cited 8 October 2019.

Ramon-Cortes C (2019b) PyCOMPSs AutoParallel Module GitHub. https://github.com/cristianrcv/pycompss-autoparallel. Cited 8 October 2019.

Ramon-Cortes C and et al (2018) Transparent Orchestration of Task-based Parallel Applications in Containers Platforms. *Journal of Grid Computing* 16(1): 137–160.

Sukumaran-Rajam A and Clauss P (2015) The Polyhedral Model of Nonlinear Loops. *ACM Trans. Archit. Code Optim.* 12(4): 48:1–48:27. DOI:10.1145/2838734.

Tejedor E and et al (2017) PyCOMPSs: Parallel computational workflows in Python. *The International Journal of High Performance Computing Applications (IJHPCA)* 31: 66–82. DOI:10.1177/1094342015594678.

The Ohio State University, Department of Computer Science and Engineering (2015) PolyBench/C: The Polyhedral Benchmark suite. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench. Cited 8 October 2019.

University of Tennesse Oak Ridge National Laboratory Numerical Algorithms Group Ltd (2017) BLAS (Basic Linear Algebra Subprograms). http://www.netlib.org/blas/. Cited 8 October 2019.

van Rossum G and Drake FL (2011) *The Python Language Reference Manual*. Network Theory Ltd. ISBN 1906966141, 9781906966140.

var der Walt S, Colbert SC and Varoquaux G (2011) The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science and Engg.* 13(2): 22–30. DOI:10.1109/MCSE.2011.37.

Vitalii Vanovschi (2019) Parallel Python Software. http://www.parallelpython.com. Cited 8 October 2019.

Zaharia M and et al (2010) Spark: Cluster Computing with Working Sets. In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. pp. 95–102.