Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

AutoParallel
A Python module for automatic parallelization and distributed execution of affine loop nests

8th Workshop on Python for High-Performance and Scientific Computing 2018

Cristián Ramón-Cortés
Ramon Amela
Jorge Ejarque
Philippe Clauss
Rosa M. Badia

Motivation

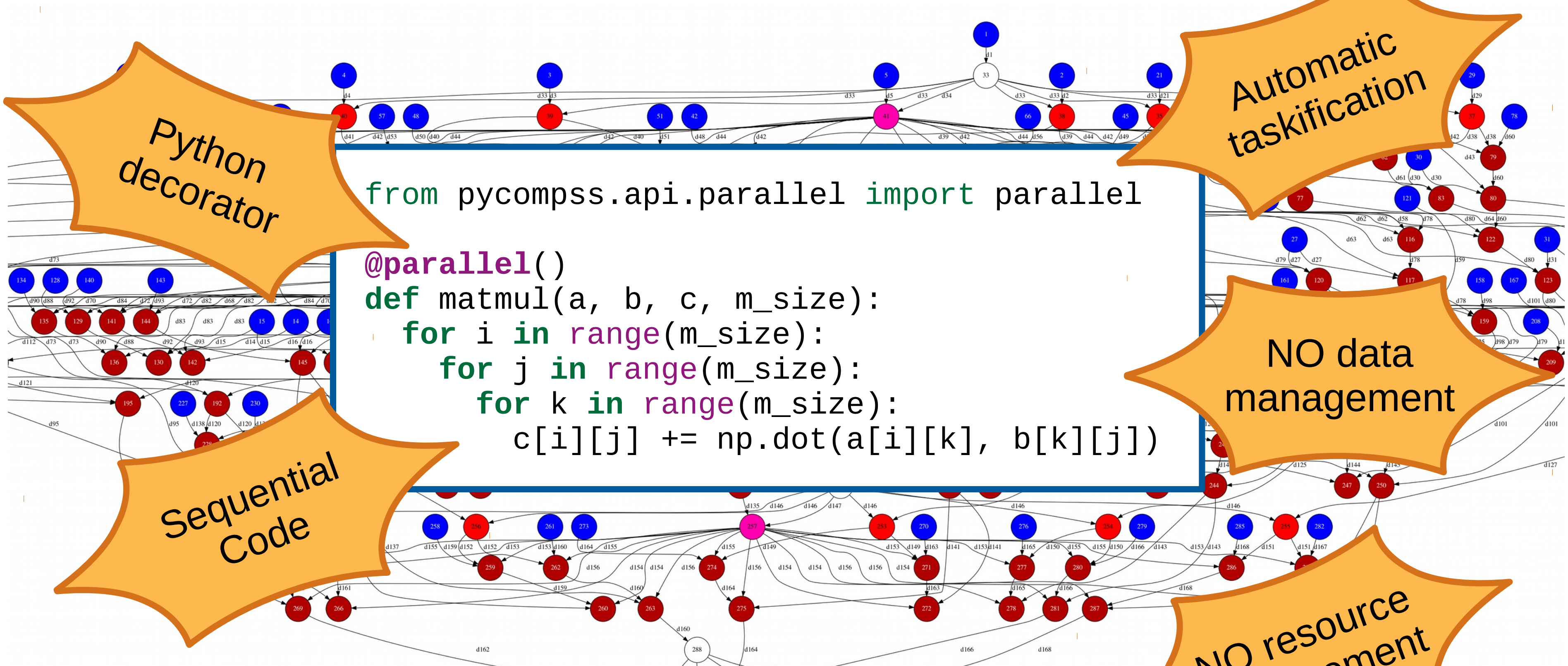
- Parallel Issues
  - Identifying parallel regions
  - Concurrency management
  - Orchestrate execution
- Distributed Issues
  - Remote execution
  - Data Transfers

Going one step further to ease the development of distributed applications

So that any field expert can scale up an application to hundreds of cores

AutoParallel Annotation

- A single Python decorator to parallelize and distributedly execute sequential code containing affine loop nests




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

@parallel()
def matmul(a, b, c, m_size):
    for i in range(m_size):
        for j in range(m_size):
            for k in range(m_size):
                c[i][j] += np.dot(a[i][k], b[k][j])
```

Outline

- Architecture
- Evaluation
- Loop taskification (Advanced feature)
- Conclusions and Future Work

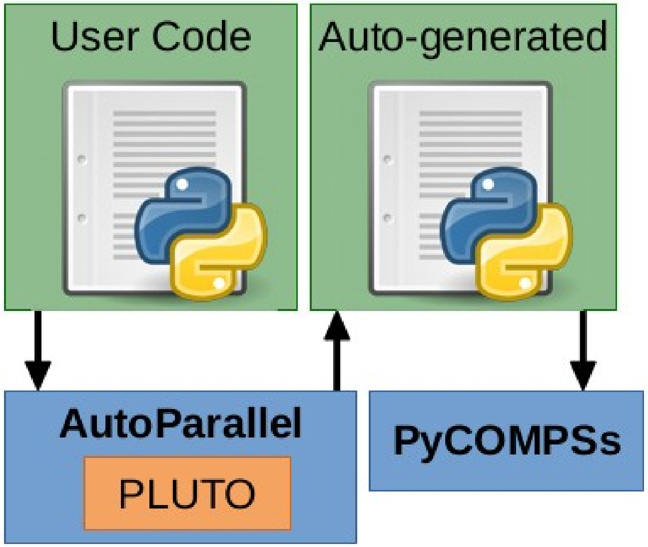
AutoParallel Architecture

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

AutoParallel Annotation

Taskification of affine loop nests at runtime

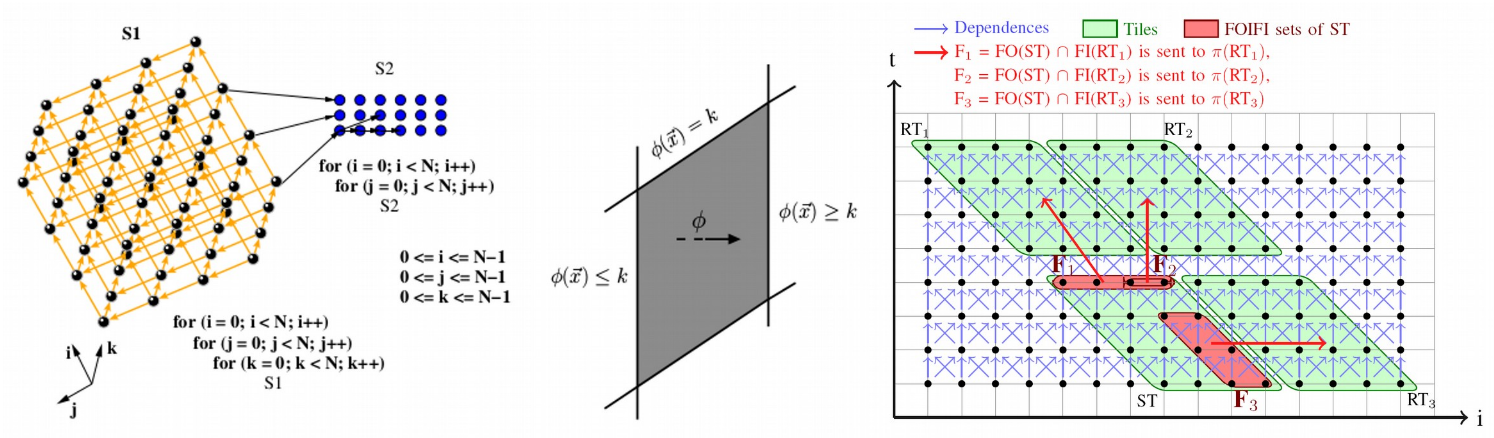
```python
@parallel()
def ep(mat, n, m, c1, c2):
  for i in range(n):
    for j in range(m):
      mat[i][j] = compute(mat[i][j], c1, c2)
```



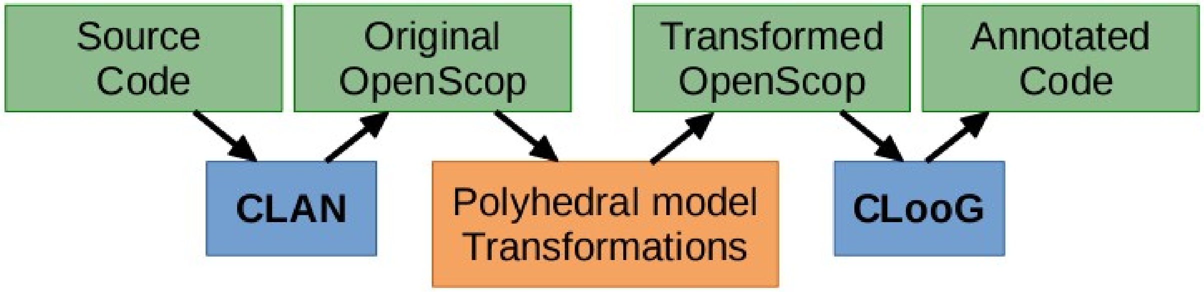

```python
# [COMPSs AutoParallel] Begin Autogenerated code
@task(var2=IN, c1=IN, c2=IN, returns=1)
def S1(var2, c1, c2):
  return compute(var2, c1, c2)

def ep(mat, n, m, c1, c2):
  if m >= 1 and n >= 1:
    lbp = 0
    ubp = m - 1
    for t1 in range(lbp, ubp + 1):
      lbv = 0
      ubv = n - 1
      for t2 in range(lbv, ubv + 1):
        mat[t2][t1] = S1(mat[t2][t1], c1, c2)
  compss_barrier()
# [COMPSs AutoParallel] End Autogenerated code
```

PLUTO Main Features

- The Polyhedral model represents the instances of the loop nests’ statements as integer points inside a polyhedron.



- PLUTO is an automatic parallelization tool based on the Polyhedral model to optimize arbitrarily nested loop sequences with affine dependencies.

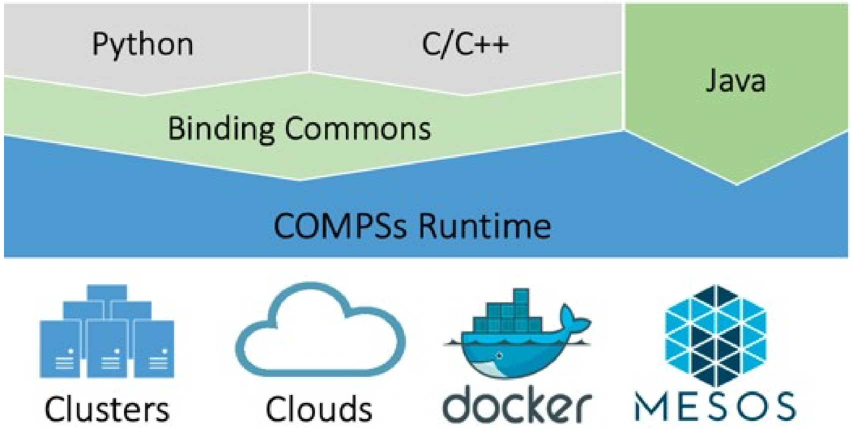


PyCOMPSs Main Features

- COMPSs is a task-based programming model which aims to ease the development of parallel applications for distributed infrastructures
- The Python binding is known as PyCOMPSs

- Based on:
  - Sequential programming
  - Selection of tasks
    - Functions (instance and class methods)
    - Task data direction

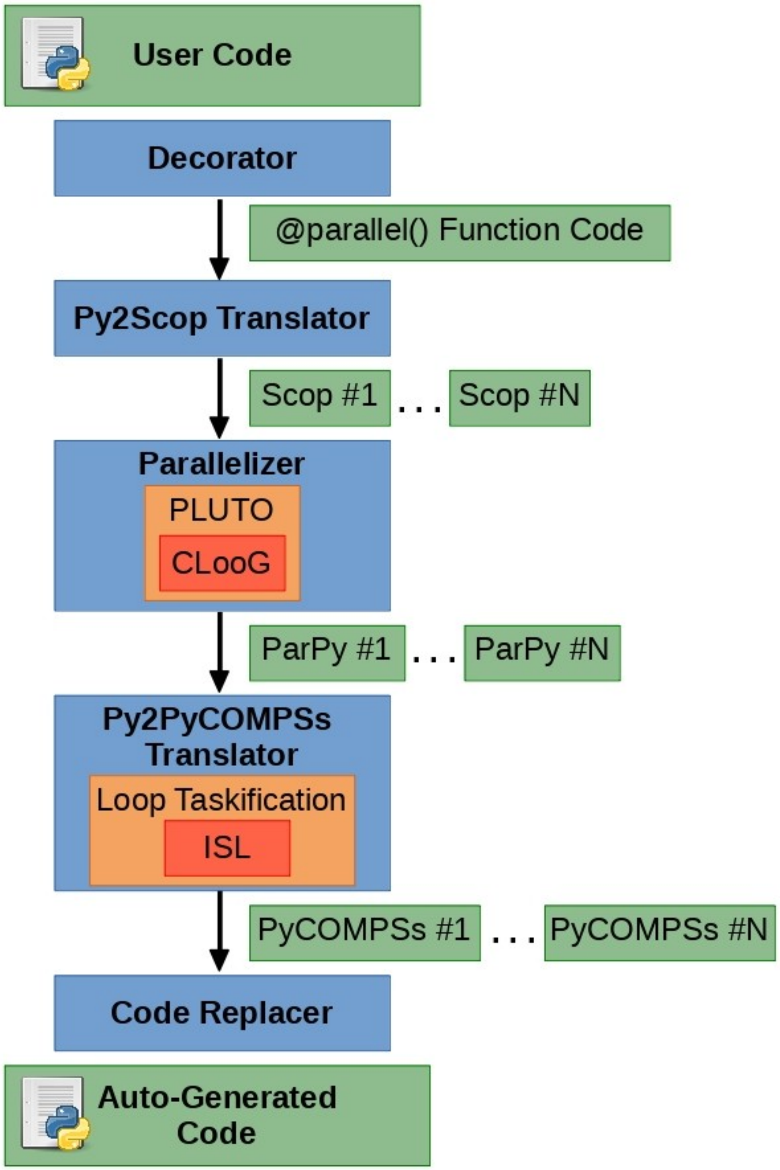
```python
@constraint(computingUnits="2")
@task(c=INOUT)
def multiply(a, b, c):
    c += a * b
```



- Same application runs on Clusters, Grids, Clouds and Containers

AutoParallel Architecture

- **Decorator:**
  - Implements the @parallel() decorator
- **Python To OpenScop Translator:**
  - Builds a Python Scop object representing each affine loop nest detected in the user function
- **Parallelizer:**
  - Returns the Python code resulting from parallelizing an OpenScop file (OpenMP syntax)
- **Python to PyCOMPSs Translator**
  - Inserts the PyCOMPSs syntax (task annotations and data synchronizations) to the annotated Python code
- **Code Replacer**
  - Replaces each loop nest in the initial user code by the autogenerated code



Evaluation

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

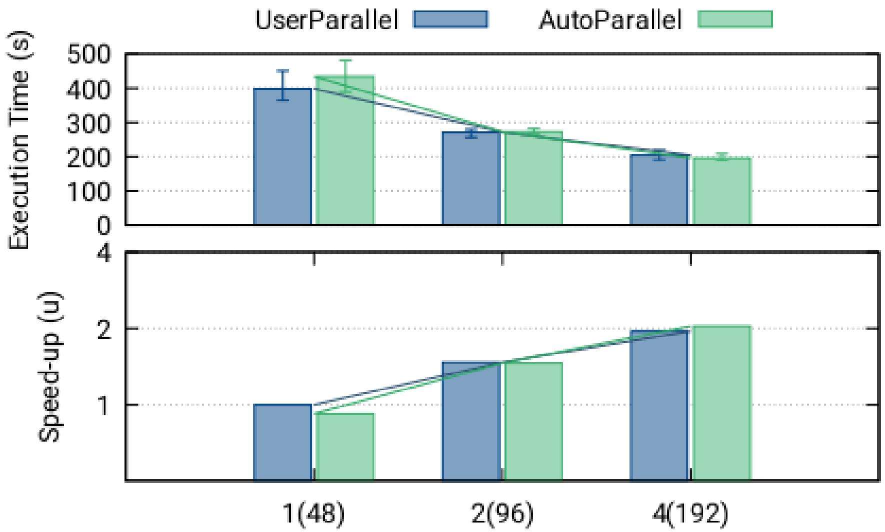
Experimentation: Blocked Applications

Cholesky

LoC	Lines Of Code	
CC	Cyclomatic Complexity	
NPath	Npath Complexity	

	Code Analysis		
	LoC	***CC***	***NPath***
User	220	26	112
Auto	274	36	14.576

	Loop Analysis		
	#Main	***#Total***	***Depth***
User	1	4	3
Auto	3	9	3



	Problem Size			Execution		
	Total Matrix Size	***#Blocks***	***Block Size***	***Task Types***	***#Tasks***	***SpeedUp @ 192 cores***
User	65.536 x 65.536	32 x 32	2048 x 2048	3	6.512	1,95
Auto				4	7.008	2,04

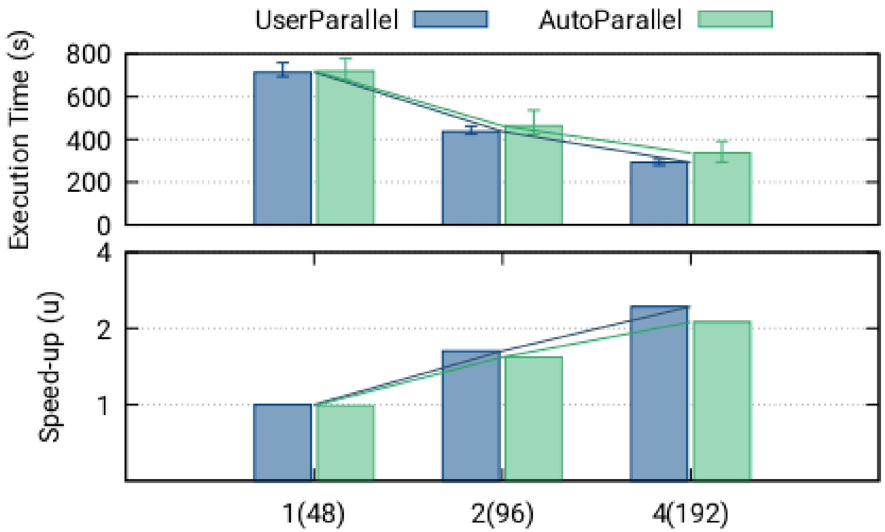
Experimentation: Blocked Applications

LU

LoC	Lines Of Code	
CC	Cyclomatic Complexity	
NPath	Npath Complexity	

	Code Analysis		
	LoC	***CC***	***NPath***
User	238	35	79.872
Auto	320	39	331.776

	Loop Analysis		
	#Main	***#Total***	***Depth***
User	2	6	3
Auto	2	6	3



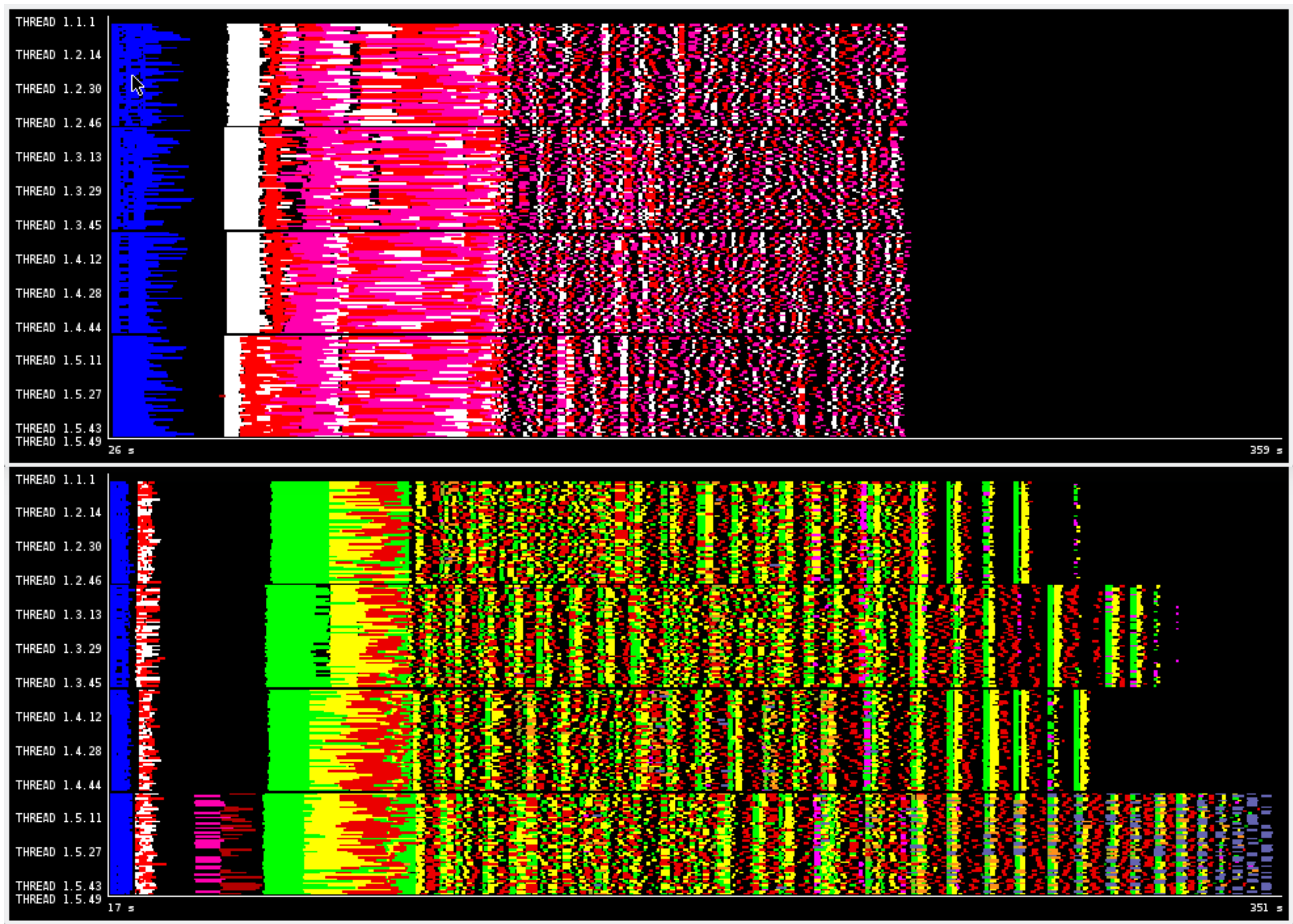
	Problem Size			Execution		
	Total Matrix Size	***#Blocks***	***Block Size***	***Task Types***	***#Tasks***	***SpeedUp @ 192 cores***
User	49.152 x 49.152	24 x 24	2048 x 2048	4	14.676	2,45
Auto				12	15.227	2,13

Experimentation: Blocked Applications

- LU: In-depth Performance Analysis
  - Paraver Trace with 4 workers (192 cores)

UserParallel

AutoParallel

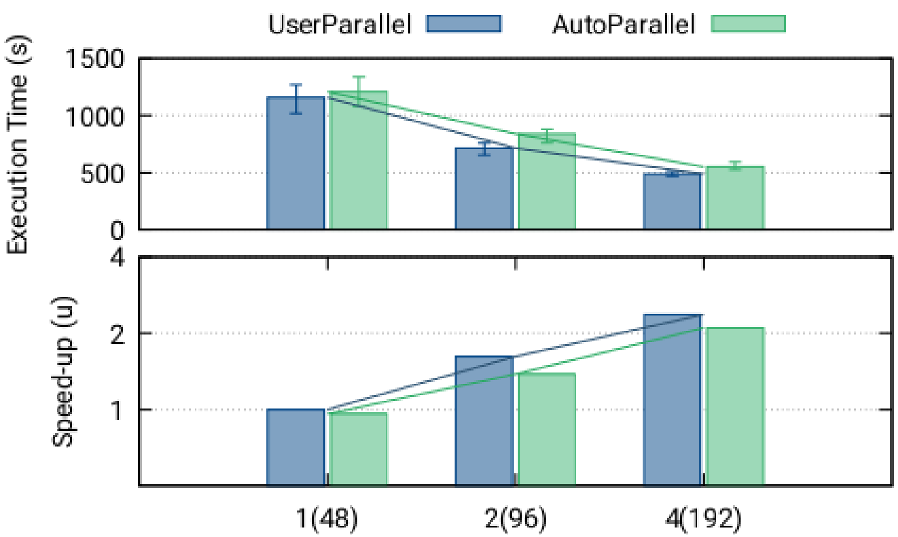


Experimentation: Blocked Applications

QR

LoC	Lines Of Code
CC	Cyclomatic Complexity
NPath	Npath Complexity

	Code Analysis		
	LoC	***CC***	***NPath***
User	303	41	168
Auto	406	43	344



	Loop Analysis		
	#Main	***#Total***	***Depth***
User	1	6	3
Auto	2	7	3

<table>
<tr><th rowspan="2"></th><th colspan="3">Problem Size</th><th colspan="3">Execution</th></tr>
<tr><th>Total Matrix Size</th><th>#Blocks</th><th>Block Size</th><th>Task Types</th><th>#Tasks</th><th>SpeedUp @ 192 cores</th></tr>
<tr><td>User</td><td rowspan="2">32.768 x 32.768</td><td rowspan="2">16 x 16</td><td rowspan="2">2048 x 2048</td><td>4</td><td>19.984</td><td>2,37</td></tr>
<tr><td>Auto</td><td>20</td><td>26.304</td><td>2,10</td></tr>
</table>

Loop Taskification: AutoParallel Advanced Feature

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

PLUTO Tiling

- Tiling a loop of given size N results in a division of the loop in N/T repeatable parts of size T

Original Loop

```
for i in range(N):
    print(i)
```

Tiled Loop

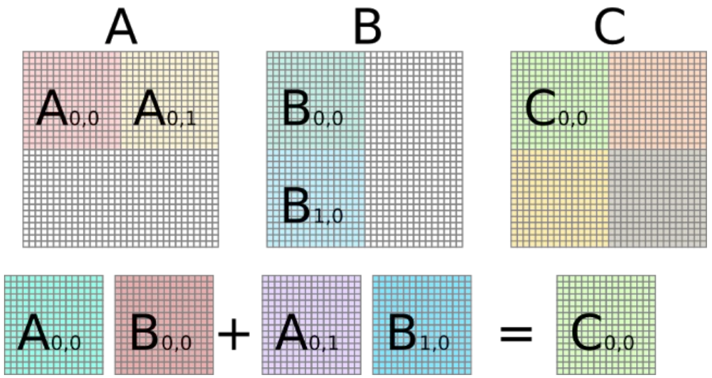
```
for i in range(N/T):
    for i in range(T):
        print(i*T + t)
```

- Tiling is designed to fit loops into the L1 or L2 caches **BUT** can be used to build data blocks to increase the tasks' granularity

AutoParallel Loop Taskification

- Convert into tasks all the loops of a certain depth
- N-dimensional arrays are divided into data blocks (chunks) for each callee and reverted after the task execution

```python
@parallel(pluto_extra_flags=["--tile"],
          taskify_loop_level=3)
def matmul(a, b, c, m_size):
  for i in range(m_size):
    for j in range(m_size):
      for k in range(m_size):
        c[i][j] += np.dot(a[i][k], b[k][j])
```




```python
@parallel(pluto_extra_flags=["--tile"],
          taskify_loop_level=3)
def matmul(a, b, c, m_size):
  for i in range(m_size/T1):
    for j in range(m_size/T2):
      for k in range(m_size/T3):
        c[..][..] = LT1(c[..][..],
                        a[..][..],
                        b[..][..])
```

```python
@task(..)
def LT1(c, a, b):
  for i' in range(T1):
    for j' in range(T2):
      for k' in range(T3):
        c[i'][j'] += np.dot(a[i'][k'],
                            b[k'][j'])
  return c
```

AutoParallel Loop Taskification

EP Generated code

- Flattening and rebuilding data chunks

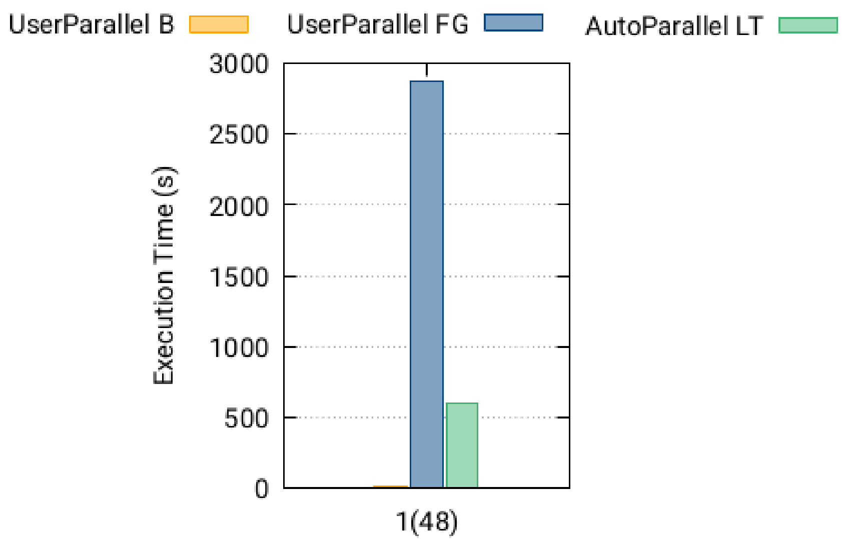
```
def ep(mat, n, m, c1, c2):
  if m >= 1 and n >= 1:
    lbp = 0
    ubp = m - 1
    for t1 in range(lbp, ubp + 1):
      lbv = 0
      ubv = n – 1
      # Chunk creation and flattening
      LT2_aux0 = [mat[t2][t1] for …]
      LT2_au = ArgUtils()
      global LT2_args_size
      LT2_flat, LT2_args_size = LT2_au.flatten(LT2_aux0)
      # Task call
      LT2_ret = LT2(lbv, ubv, c1, c2, *LT2_flat)
      # Rebuild and re-assign
      LT2_aux_0, = LT2_au.rebuild(LT2_ret)
      ...
  compss_barrier()
```

```
@task(lbv=IN, ubv=IN, c1=IN, c2=IN,
      returns=”LT2_args_size”)
def LT2(lbv, ubv, c1, c2, *args):
  global LT2_args_size
  var1, = ArgUtils.rebuild_args(args)
  for t2 in range(0, ubv + 1 – lbv):
    var1[t2] = S1_no_task(var1[t2],
                          c1, c2)
  return ArgUtils.flatten_args(var1)

def S1_no_task(var2, c1, c2):
  return compute(var2, c1, c2)
```

Experimentation: Fine-grain Applications

GEMM

	Code Analysis			Loop Analysis			Task Types
	LoC	*CC*	*NPath*	*#Main*	*#Total*	*Depth*	
User B	189	22	112	2	5	3	2
User FG	194	22	112	1	4	3	2
Auto LT	382	133	360064	2	4	3	4



Conclusions and Future Work

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Conclusions and Future Work

- AutoParallel goes one step further in easing the development of distributed applications
  - It is a Python module to automatically parallelize affine loop nests and execute them in distributed infrastructures
  - The evaluation shows that the automatically generated codes for the Cholesky, LU, and QR applications can achieve the same performance than the manually parallelized versions

- Next steps
  - Loop Taskification provides an automatic way to create blocks from sequential applications, but its performance is still far from acceptable.
    - Research on how to simplify the chunk accesses from the AutoParallel module.
    - Extend PyCOMPSs to support collection objects (e.g., lists)
  - AutoParallel could be integrated with different tools similar to PLUTO to support a larger scope of loop nests.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

Thank you

Live demos at BSC booth #2038

Tuesday	4:40 pm
Wednesday	2:00 pm
Thursday	11:20 am

cristianrcv/pycompss-autoparallel

http://compss.bsc.es/

cristian.ramoncortes@bsc.es