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
A single Python decorator to parallelize and distribute sequentially 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])
```

**AutoParallel Annotation**

**Python decorator**

**Sequential Code**

**Automatic taskification**

**NO data management**

**NO resource management**

*Grid*  *Cluster*  *Cloud*  *Docker*
Outline

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

Taskification of affine loop nests at runtime

@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)

# [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
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
## Experimentation: Blocked Applications

### Cholesky

<table>
<thead>
<tr>
<th>Code Analysis</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LoC</strong></td>
<td><strong>CC</strong></td>
<td><strong>NPath</strong></td>
<td></td>
</tr>
<tr>
<td>User</td>
<td>220</td>
<td>26</td>
<td>112</td>
</tr>
<tr>
<td>Auto</td>
<td>274</td>
<td>36</td>
<td>14.576</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Loop Analysis</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>#Main</strong></td>
<td><strong>#Total</strong></td>
<td><strong>Depth</strong></td>
<td></td>
</tr>
<tr>
<td>User</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Auto</td>
<td>3</td>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total Matrix Size</strong></td>
<td><strong>#Blocks</strong></td>
</tr>
<tr>
<td>User</td>
<td>65.536 x 65.536</td>
</tr>
<tr>
<td>Auto</td>
<td></td>
</tr>
</tbody>
</table>
Experimentation: Blocked Applications

**LU**

### Code Analysis

<table>
<thead>
<tr>
<th></th>
<th>LoC</th>
<th>CC</th>
<th>NPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>238</td>
<td>35</td>
<td>79.872</td>
</tr>
<tr>
<td>Auto</td>
<td>320</td>
<td>39</td>
<td>331.776</td>
</tr>
</tbody>
</table>

### Loop Analysis

<table>
<thead>
<tr>
<th></th>
<th>#Main</th>
<th>#Total</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Auto</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

### Problem Size

<table>
<thead>
<tr>
<th></th>
<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>
</thead>
<tbody>
<tr>
<td>User</td>
<td>49.152 x 49.152</td>
<td>24 x 24</td>
<td>2048 x 2048</td>
<td>4</td>
<td>14.676</td>
<td>2.45</td>
</tr>
<tr>
<td>Auto</td>
<td>2048 x 2048</td>
<td></td>
<td></td>
<td>12</td>
<td>15.227</td>
<td>2.13</td>
</tr>
</tbody>
</table>
Experimentation: Blocked Applications

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

### Code Analysis

<table>
<thead>
<tr>
<th></th>
<th>LoC</th>
<th>CC</th>
<th>NPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>303</td>
<td>41</td>
<td>168</td>
</tr>
<tr>
<td>Auto</td>
<td>406</td>
<td>43</td>
<td>344</td>
</tr>
</tbody>
</table>

### Loop Analysis

<table>
<thead>
<tr>
<th></th>
<th>#Main</th>
<th>#Total</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>1</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Auto</td>
<td>2</td>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>

### Problem Size and Execution

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total Matrix Size</td>
</tr>
<tr>
<td>User</td>
<td>32.768 x 32.768</td>
</tr>
<tr>
<td>Auto</td>
<td>32.768 x 32.768</td>
</tr>
</tbody>
</table>
Loop Taskification: AutoParallel Advanced Feature
Tiling a loop of given size $N$ results in a division of the loop in $N/T$ repeatable parts of size $T$

**Original Loop**

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

**Tiled Loop**

```python
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[i][j] += np.dot(a[i][k], b[k][j])
```

```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

```python
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)
```
### GEMM

#### Code Analysis

<table>
<thead>
<tr>
<th></th>
<th>LoC</th>
<th>CC</th>
<th>NPath</th>
<th>#Main</th>
<th>#Total</th>
<th>Depth</th>
<th>Task Types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>User B</strong></td>
<td>189</td>
<td>22</td>
<td>112</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td><strong>User FG</strong></td>
<td>194</td>
<td>22</td>
<td>112</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td><strong>Auto LT</strong></td>
<td>382</td>
<td>133</td>
<td>360064</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

#### Loop Analysis

<table>
<thead>
<tr>
<th></th>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserParallel B</td>
<td>1(48)</td>
</tr>
<tr>
<td>UserParallel FG</td>
<td></td>
</tr>
<tr>
<td>AutoParallel LT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3000</td>
</tr>
<tr>
<td></td>
<td>2500</td>
</tr>
<tr>
<td></td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td>1500</td>
</tr>
<tr>
<td></td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>500</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

### Experimentation: Fine-grain Applications
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.
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