CharmPy: Parallel Programming with Python Objects

Juan Galvez
April 11, 2018
16th Annual Workshop on Charm++ and its Applications
What is CharmPy?

- Parallel/distributed programming framework for Python
- Charm++ programming model (Charm++ for Python)
- High-level, general purpose
- Runs on top of Charm++ runtime (C++)
- Good runtime performance
- Adaptive runtime features: asynchronous remote method invocation, dynamic load balancing, automatic communication/computation overlap
Why CharmPy?

- Python+Charmpy easy to learn/use, many productivity benefits
- Bring Charm++ to Python community
  - No high-level & fast & highly-scalable parallel frameworks for Python
- Benefit from Python software stack
  - Python widely used for data analytics, machine learning
  - Opportunity to bring data and HPC closer
- Cons?
  - Potentially, performance, BUT performance can be similar to C++
Charmpy Python-derived benefits

- Productivity (high-level, less lines of code, easy to debug)
- Automatic memory management
- Automatic object serialization
  - No need to define serialization (PUP) routines
  - Can customize serialization if needed
- Easy access to Python software libraries (numpy, pandas, scikit-learn, TensorFlow, etc)
Charmpy-specific features

• Simplifies Charm++ programming
  – Much simpler, more intuitive API

• No specialized languages, preprocessing or compilation
  – Using reflection/introspection
  – Everything can be expressed in Python
  – No interface (ci) files!
```python
#hello_world.py
from charmpy import charm, Chare, Group

class Hello(Chare):
    def sayHi(self, vals):
        print('Hello from PE', charm.myPe(), 'vals=', vals)
        self.contribute(None, None, self.thisProxy[0].done)

    def done(self): charm.exit()

def main(args):
    g = Group(Hello)  # create a Group of Hello chares
    g.sayHi([1, 2.33, 'hi'])

charm.start(entry=main)
```
Run Hello World

```bash
$ ./charmrun +p4 /usr/bin/python3 hello_world.py
# similarly on a supercomputer with aprun/srun/...

Hello from PE 0 vals= [1, 2.33, 'hi']
Hello from PE 3 vals= [1, 2.33, 'hi']
Hello from PE 1 vals= [1, 2.33, 'hi']
Hello from PE 2 vals= [1, 2.33, 'hi']
```
Charmpy components

1. **Python application**
   - import charmpy

2. **charmpy module**
   - cython

3. **charmlib interface layer**
   - ctypes
   - cffi
   - cython

4. **Charm++ shared library**
   - (libcharm.so)

Other Python libraries/technologies:
- numpy, numba, pandas, matplotlib, scikit-learn, TensorFlow, ...

C / C++ / Fortran / OpenMP
What about performance?

- Many (compiled) parallel programming languages proposed over the years for HPC
- **Use Python in same way**: high-level language driving machine-optimized compiled code
  - Numpy (high-level arrays/matrices API, native implementation)
  - Numba (JIT compiles Python “math/array” code)
  - Cython (compile generic Python to C)
Numba

- Compiles Python to native machine using LLVM compiler
  - Good for loops and numpy array code

```python
@numba.jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return result

a = arange(9).reshape(3,3)
print(sum2d(a))
```
(from http://numba.pydata.org)
Numba

• Interesting feature:
  – Input parameters that are normally variables can be compiled as constants thanks to JIT compilation

```python
@numba.jit
def compute(arr, ...):
    for x in range(block_size_x):
        for y in range(block_size_y):
            arr[x,y] = ...
```

• Can write CUDA kernels

Values can be supplied at launch, but be compiled as constants
Chares are distributed Python objects

- Remote methods invoked like regular Python objects, via proxy:
  ```python
  obj_proxy.doWork(x, y)
  ```
- Objects are migratable (handled by Charm++ runtime)
- Method invocation asynchronous in general (good for performance)
- Can also do:
  ```python
  ret = obj_proxy.getVal(block=True)
  ```
  - Caller gets value returned by remote method
  - Entry method on which call is made needs to be marked as @threaded (runtime will inform)
Distributed collections (Groups, Arrays)

```python
group = Group(MyChare)  # one instance per PE
group

array = Array(MyChare, (100,100))  # 2D array, 100x100
    # instances

array.work(x,y,z)  # invoke method on all objects in array

array[3,10].work(x,y,z)  # invoke method on object with index (3,10)
```
Reductions

- Reduction (e.g. sum) by elements in a collection:

```python
def work(self, x, y, z):
    A = numpy.arange(100)
    self.contribute(A, Reducer.sum, obj.collectResults)
```

- Easy to define custom reducer functions. Example:

  - ```python
def mysum(contributions):
      return sum(contributions)
  ```
  - ```python
      self.contribute(A, Reducer.mysum, obj.collectResult)
  ```
Benchmark using stencil3d

- In `examples/stencil3d`, ported from Charm++
- Stencil code, 3D array decomposed into shines
- Full Python application, array/math sections JIT compiled with Numba
- Cori KNL 2 nodes, strong scaling from 8 to 128 cores
stencil3d results on Cori KNL

stencil3d on Cori KNL 2 nodes, strong scaling

- **Charm++**
- **Charmpy**

<table>
<thead>
<tr>
<th># cores</th>
<th>time per iteration (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>+1%</td>
</tr>
<tr>
<td>16</td>
<td>+3%</td>
</tr>
<tr>
<td>32</td>
<td>+5%</td>
</tr>
<tr>
<td>64</td>
<td>+8%</td>
</tr>
<tr>
<td>128</td>
<td>+29%</td>
</tr>
</tbody>
</table>
Evolution of performance

stencil3d, relative difference to Charm++

<table>
<thead>
<tr>
<th>Core #</th>
<th>ctypes</th>
<th>cffi</th>
<th>cython</th>
<th>Charm++</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Benchmark using LeanMD

• MD mini-app for Charm++ (http://charmplusplus.org/miniApps/#leanmd)
  – Simulates the behavior of atoms based on the Lennard-Jones potential
  – Computation mimics the short-range non-bonded force calculation in NAMD
  – 3D space consisting of atoms decomposed into cells
  – In each iteration, force calculations done for all pairs of atoms within the cutoff distance

• Ported to Charmpy, full Python application. Physics code and other numerical code JIT compiled with Numba
LeanMD results on Blue Waters

Performance on Blue Waters (8 million particles)

Avg difference is 19%
(results not based on latest Charmpy version)
Serialization (aka pickling)

- Most Python types, including custom types, can be pickled
- Can customize pickling with `__getstate__` and `__setstate__` methods
- pickle module implemented in C, recent versions are pretty fast (for built-in types)
  - Pickling custom objects not recommended in critical path
- Charmpy bypasses pickling for certain types like numpy arrays
Shared memory parallelism

- In the Python interpreter, **NO**
  - CPython (most common Python implementation) still can’t run multiple threads *concurrently*

- Outside the interpreter, **YES**
  - Numpy internally runs compiled code, can use multiple threads (Intel Python + numpy seems to be very good at this)
  - Access external OpenMP code from Python
  - Numba parallel loops
Summary

- Easy way to write parallel programs based on Charm++ model
- Good runtime performance
  - Critical sections of Charmpy runtime in C with Cython
  - Most of the runtime is C++
- High performance using NumPy, Numba, Cython, interacting with native code
- Easy access to Python libraries, like SciPy and PyData stacks
Thank you!

- More resources:
- Documentation and tutorial at http://charmpy.readthedocs.io
- Examples in project repo: https://github.com/UIUC-PPL/charmpy