Charm4py: Parallel Programming with Python and Charm++

PARALLEL PROGRAMMING LABORATORY

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

What is Charm4py?

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

Why Charm4py?

- Python + Charm4py 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
- Performance can be similar to C/C++ using the right techniques

Charm4py Python-derived benefits

- Productivity (high-level, fewer lines of code, easy to debug)
- Automatic memory management
- Automatic serialization
  - No need to define serialization (PUP) routines
  - Can customize serialization of Chares if needed
- Easy access to Python software libraries (Numpy, pandas, scikit-learn, TensorFlow, etc.)

Charm4py-specific features

- Simplifies Charm++ programming
  - Simpler, more intuitive API
- No specialized languages, preprocessing or compilation
  - Everything can be expressed in Python
  - **No interface (ci) files!**

Hello World (complete example)

```python
#hello_world.py
from charm4py import charm, Chare, Group

class Hello(Chare):
    def sayHi(self, values):
        print('Hello from PE', charm.myPe(), 'vals=', values)
        self.contribute(None, None, charm.thisProxy.exit)

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

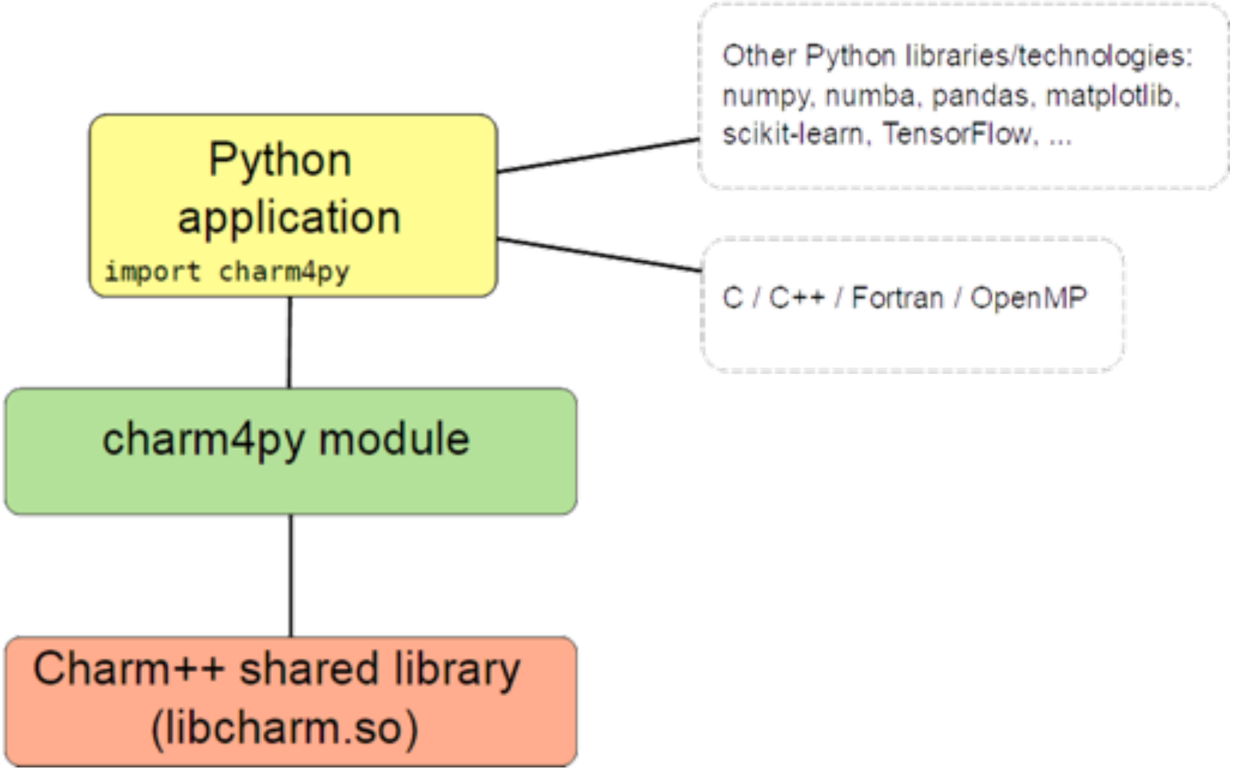
charm.start(main)
```

Running Hello World

```
$ ./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']
```

Charm4py components



What about performance?

- Many (compiled) parallel programming languages proposed over the years for HPC
- Can use Python in the 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
  - Can also write CUDA kernels

(from http://numba.pydata.org)

```
@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))
```

Shared memory parallelism

- Inside the Python interpreter, **NO**
  - CPython (most common Python implementation) can't run multiple threads *concurrently* (Global Interpreter Lock)
- 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
  - Cython

Chares are distributed Python objects

- Remote methods (aka entry methods) invoked like regular Python objects, via proxy: `obj_proxy.doWork(x, y)`
- Objects are migratable (handled by Charm++ runtime)
- Method invocation asynchronous (good for performance)
- Can obtain a *future* when invoking remote methods:
  - ```
    future = obj_proxy.getVal(ret=True)
    ... do work ...
    val = future.get()  # block until value received
    ```
    ```

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
- Charm4py bypasses pickling for certain types like Numpy arrays

Creating chares

```
class MyChare(Chare):
    def __init__(self, x):
        self.x = x
    def work(self, param1, param2, param3):
        ...

def main(args):
    # create single chare of type MyChare on PE 1
    obj_proxy = Chare(MyChare, args=[1], onPE=1)
    # create Group (one instance per PE)
    group_proxy = Group(MyChare, args=[1])
```

Creating chares (cont.)

```
def main(args):
    ...
    # create 2D array, 100x100 instances of MyChare
    array_proxy = Array(MyChare, (100,100), args=[3])
    # invoke method on all members
    array_proxy.work(x, y, z)
    # invoke method on object with index (3,10)
    array_proxy[3,10].work(x, y, z)
```

Futures

- Threaded entry methods run in their own thread
    - @threaded
      def myThreadedEntryMethod(self, …):
    - Main function (or mainchare constructor) is threaded by default
- Threaded entry methods can use futures to wait for a result or for completion of a (distributed) process
- While a thread is blocked, other entry methods in the same process (of the same or different chares) continue to be scheduled and executed

Futures (cont.)

```
@threaded
def someEntryMethod(self, ...):
    a1 = Array(MyChare, 100) # create array of 100 elems
    a2 = Array(MyChare, 20) # create array of 20 elems
    charm.awaitCreation(a1, a2) # wait for creation
    f1 = a1[0].calculateValue(ret=True)
    f2 = a2[0].calculateValue(ret=True)
    a2.initialize(ret=True).get() # wait for broadcast completion
    val1 = f1.get()
    val2 = f2.get()
    f3 = charm.createFuture()
    a1.work(f3)
    f3.get()  # wait for completion
```

Blocking collectives

- Blocking collectives are available for threaded entry methods (use futures internally):

```
@threaded
def someEntryMethod(self, ...):
    # wait for elements in my collection to reach barrier
    charm.barrier(self)
    # blocking allReduce among members of collection
    result = charm.allReduce(data, reducer, self)
```

Reductions

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

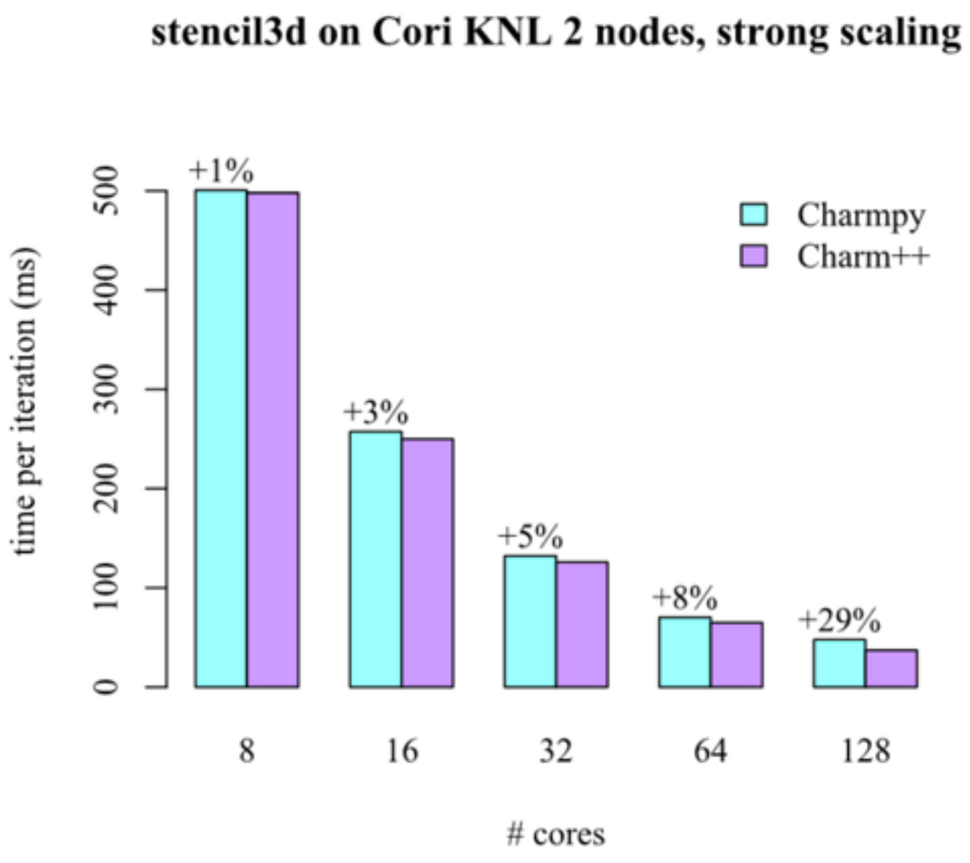
```
def work(self, x, y, z):
    A = numpy.arange(100)
    self.contribute(A, Reducer.sum, obj_proxy.collectResults)
```

- Target of reduction can be an entry method or a future
- Easy to define custom reducer functions. Example:
  - `def mysum(contributions): return sum(contributions)`
  - `self.contribute(A, Reducer.mysum, obj.collectResult)`

Benchmark using stencil3d

- In examples/stencil3d, ported from Charm++
- Stencil code, 3D array decomposed into chares
- 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

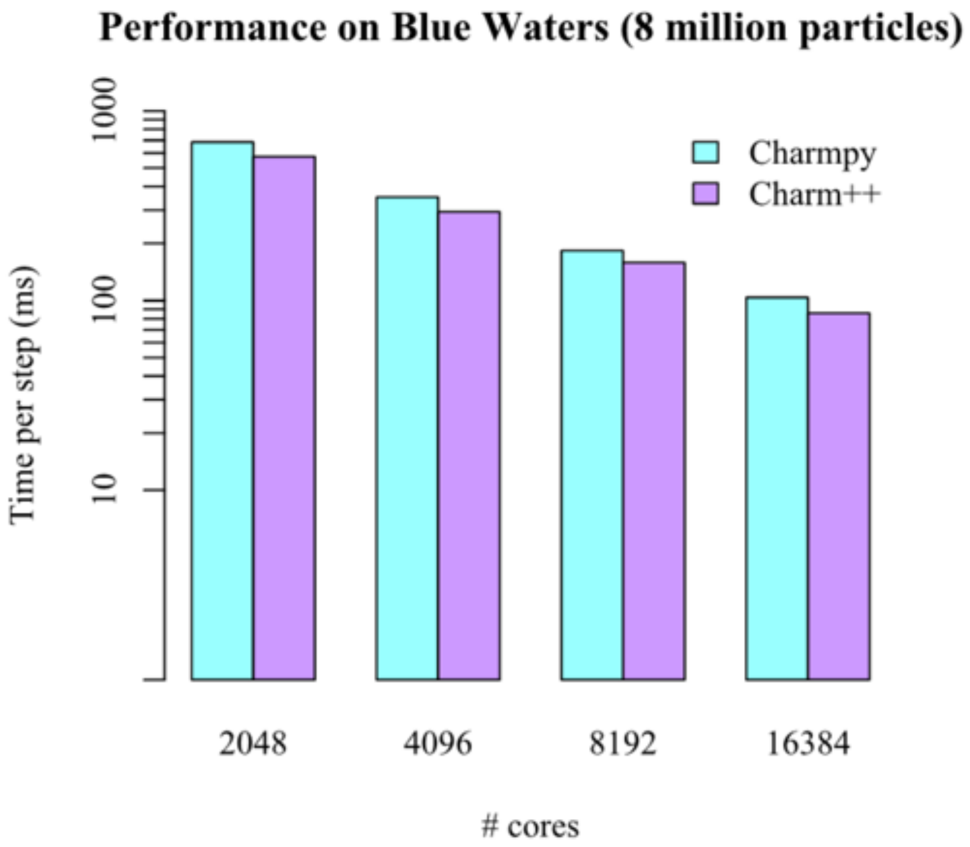


(results not based on latest Charm4py version)

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 Charm4py, full Python application. Physics code and other numerical code JIT compiled with Numba

LeanMD results on Blue Waters



Avg difference is 19%

(results not based on latest Charm4py version)

Experimental features

- Interactive mode
  - Launches an interactive Python shell where user can define new chares, create them, invoke remote methods, etc.
  - Currently for (multi-process) single node
- Distributed pool of workers for task scheduling:

```
def fib(n):
    if n < 2: return n
    return sum(charm.pool.map(fib, [n-1, n-2],
                   allow_nested=True))

def main(args):
    result = fib(33)
```

Summary

- Easy way to write parallel programs based on Charm++ model
- Good runtime performance
  - Critical sections of Charm4py 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://charm4py.readthedocs.io
- Source code and examples at: https://github.com/UIUC-PPL/charm4py