Charm4py Documentation

Release 1.0.0

Juan Galvez

Sep 09, 2019

Getting Started

Charm4py is a distributed computing and parallel programming framework for Python, for the productive development of fast, parallel and scalable applications. It is built on top of Charm++, an adaptive runtime system that has seen extensive use in the scientific and high-performance computing (HPC) communities across many disciplines, and has been used to develop applications like NAMD that run on a wide range of devices: from small multi-core devices up to the largest supercomputers.

CHAPTER 1

Features

- **Actor model**: Charm4py employs a simple and powerful actor model for concurrency and parallelism. Applications are composed of distributed Python objects; objects can invoke methods of any other objects in the system, including those on other hosts. This happens via message passing, and works in the same way regardless of the location of source and destination objects.
- **Asynchronous**: every operation, including remote method invocation, is executed asynchronously. This contributes to better resource utilization and overlap of computation and communication.
- **Concurrency**: multiple concurrency features are seamlessly integrated into the actor model, including couroutines, channels and futures, that facilitate writing in direct or sequential style. See the *Introduction* for a quick overview.
- **Speed**: The core Charm++ library is implemented in C/C++, making runtime overhead very low. A Cython module offers efficient access to the library. Charm++ has been used in high-performance computing for many years, with applications scaling to the world's top supercomputers.
- **Load balancing** of persistent objects: distributed objects can be migrated by the runtime dynamically to balance computational load, in a way that is transparent to applications.
- **Parallel tasks** using a distributed pool of workers (which works across multiple hosts). Tasks are Python functions and coroutines. The framework supports efficient nested parallelism (tasks can create and wait for other tasks). Among the operations supported are large-scale parallel map (akin to Python multiprocessing's map), and the ability to spawn individual tasks, which can be used to easily implement parallel state space search or similar algorithms. The runtime decides where to launch tasks and balances them across processes.
- **High-performance communication**: Charm4py offers a choice of multiple high-performance communication layers (when manually building the Charm++ library), including MPI as well as native layers for many high-performance interconnects like Cray GNI, UCX, Intel OFI and IBM PAMI, with features like shared memory and RDMA.

CHAPTER 2

Introduction

Charm4py's programming model is based on an actor model. Distributed objects in Charm4py are called *Chares* (pronounced char). A chare is essentially a Python object in the OOP sense with its own attributes (data) and methods. A chare lives on one process, and some chares can migrate between processes (e.g. for dynamic load balancing). Chares can call methods of any other chares in the system via remote method invocation, with the same syntax as calling regular Python methods. The runtime automatically takes care of location management and uses the most efficient technique for method invocation and message passing. Parallelism is achieved by having chares distributed across processes/cores.

You can create as many collections of distributed objects as you want, of the same or different types. There can be multiple chares on one process, each executing one or multiple tasks. Having many chares per core can help the runtime maximize resource utilization, dynamically balance load and overlap communication and computation.

In addition, Charm4py supports the following features to facilitate expression of concurrency: coroutines, channels and futures. These are seamlessly integrated into the actor model.

We will show some simple examples now to quickly illustrate these concepts. For a more step-by-step tutorial, you can check the *Tutorial* which can be done in an interactive session.

2.1 Actor model

Chares are defined as regular Python classes that are a subclass of `Chare`:

```
from charm4py import charm, Chare, Array

# Define my own chare type A (instances will be distributed objects)
class A(Chare):

    def start(self):
        # call method 'sayHi' of element 1 in my Array
        self.thisProxy[1].sayHi('hello world')

    def sayHi(self, message):
```

(continues on next page)

(continued from previous page)

```
        print(message, 'on process', charm.myPe())
        exit()

def main(args):
    # create a distributed Array of 2 objects of type A
    array_proxy = Array(A, 2)
    # call method 'start' of element 0 of the Array
    array_proxy[0].start()

# start the Charm runtime. after initialization, the runtime will call
# function 'main' on the first process
charm.start(main)
```

One important thing to note here is that in Charm4py *every remote method invocation is asynchronous*. This allows the runtime to maximize resource efficiency and overlap communication and computation. This also means that calls will return immediately. You can, however, request a future when calling remote methods, and use the future to suspend the current coroutine until the remote method completes, or to obtain a return value (more on this below).

2.2 Coroutines

Chare methods can act as coroutines, which simply means that they can suspend their execution to wait for events/messages, and continue where they left off when the event arrives. This can allow writing significant parts of your code in direct or sequential style. Simply decorate a method with `@coro` to allow it to work as a coroutine. When a coroutine suspends, the runtime is free to schedule other work on the same process, even for the same chare.

Coroutines are typically used in conjunction with channels and futures (described below).

2.3 Channels

Channels establish streamed connections between chares (currently one-to-one). Messages can be sent/received to/from the channel using the methods `send()` and `recv()`. The following example uses Channels and coroutines:

```
from charm4py import charm, Chare, Array, coro, Channel

class A(Chare):

    @coro
    def start(self):
        if self.thisIndex == (0,):
            # I am element 0, establish a Channel with element 1 of my Array
            ch = Channel(self, remote=self.thisProxy[1])
            # send msg on the channel (this is asynchronous)
            ch.send('hello world')
        else:
            # I am element 1, establish a Channel with element 0 of my Array
            ch = Channel(self, remote=self.thisProxy[0])
            # receive msg from the channel. coroutine suspends until the msg arrives
            print(ch.recv())
            exit()

def main(args):
```

(continues on next page)

(continued from previous page)

```
    a = Array(A, 2)
    # call method 'start' of every element of the array (this is a broadcast)
    a.start()

charm.start(main)
```

Tip: Coroutine methods are currently implemented using greenlets, which are very lightweight. The amount of overhead they add is tiny, so don't hesitate to use them where appropiate. Also note that the runtime will tell you if `@coro` is needed.

2.4 Futures

Coroutines can also create futures and use them to wait for certain events/messages. A future can be sent to other chares in the system, and any chare can send a value to the future, which will resume the coroutine that was waiting on it. For example:

```
from charm4py import charm, Chare, Array, coro, Channel, Future

class A(Chare):

    @coro
    def start(self, done):
        neighbor = self.thisProxy[(self.thisIndex[0] + 1) % 2]
        # establish a channel with my neighbor
        ch = Channel(self, remote=neighbor)
        # each chare sends and receives a msg to/from its neighbor for 10 steps
        for i in range(10):
            ch.send(i)
            assert ch.recv() == i
        if self.thisIndex == (0,):
            # signal the future that we are done
            done()

def main(args):
    a = Array(A, 2)
    # create a Future
    done = Future()
    # call start method on both elements (broadcast), passing the future
    a.start(done)
    # ... do work ...
    # 'get' suspends the coroutine until the future receives a value
    # (note that the main function is always a coroutine)
    done.get()
    exit()

charm.start(main)
```

2.5 Awaitable remote method calls

As mentioned above, you can also obtain a future when invoking a remote method of any chare. This is done by using the keywords `awaitable=True` and `ret=True` when calling the method. The former specifies that the call is awaitable and allows waiting for completion. The latter specifies that the caller wants to receive the return value(s). Note that `ret=True` automatically implies that the call is awaitable (a return value can only be received after the call has completed).

Example:

```
from charm4py import charm, Chare, Array

class A(Chare):

    def work(self):
        result = # ... do some work ...
        return result

def main(args):
    a = Array(A, 2)
    future = a[1].work(ret=True)
    # ... can do other stuff while the remote chare works ...
    # query future now. will suspend 'main' if the value has not arrived yet
    value = future.get()
    print('Result is', value)
    exit()

charm.start(main)
```

Caution: For broadcasts, `ret=True` will cause a list of return values to be sent to the caller. This is more expensive than simply waiting for completion of the broadcast with `awaitable=True`, and can also result in very long lists of return values if you are broadcasting to thousands of chares. In summary, only use `ret=True` for broadcasts if a list of return values is what you want.

CHAPTER 3

Install

Charm4py runs on Linux, macOS, Windows, Raspberry Pi, and a wide variety of clusters and supercomputer environments (including many supercomputers in the TOP500).

Charm4py runs on Python 2.7 and 3.3+. Python 3 is *highly* recommended for best performance and for continued support. Charm4py has been tested with the following Python implementations: CPython (most common implementation) and PyPy.

3.1 pip

To install on regular Linux, macOS and Windows machines, do:

```
$ pip3 install charm4py
```

Note: This option uses Charm++'s TCP layer as the communication layer. If you want a faster communication layer (e.g. MPI), see "Install from source" below.

pip >= 8.0 is recommended to simplify the install and avoid building Charm4py or any dependencies from sources.

Note that a 64-bit version of Python is required to install and run Charm4py.

3.2 Install from source

Note: This is not required if installing from a binary wheel with pip.

Prerequisites:

- CPython: numpy, greenlet and cython (`pip3 install 'numpy>=1.10.0' cython greenlet`)

- PyPy: none

To build the latest *stable* release, do:

```
$ pip3 install [--mpi] charm4py --no-binary charm4py
```

Or download the source distribution from PyPI, uncompress and run `python3 setup.py install [--mpi]`.

The optional flag `--mpi`, when enabled, will build the Charm++ library with the MPI communication layer (MPI headers and libraries need to be installed on the system).

To build the latest *development* version, download Charm4py and Charm++ source code and run setup:

```
$ git clone https://github.com/UIUC-PPL/charm4py
$ cd charm4py
$ git clone https://github.com/UIUC-PPL/charm charm_src/charm
$ python3 setup.py install [--mpi]
```

Note: The TCP layer (selected by default) will work on desktop, servers and small clusters. The MPI layer is faster and should work on most systems including large clusters and supercomputers. Charm++ however also has support for specialized network layers like uGNI and UCX. To use these, you have to manually build the Charm++ library (see below).

3.3 Manually building the Charm++ shared library

This is needed when building Charm++ for specialized machine/network layers other than TCP and MPI (e.g. Cray XC/XE).

Before running `python3 setup.py` in the steps above, enter the Charm++ source code directory (`charm_src/charm`), and manually build the Charm++ library. The build command syntax is:

```
$ ./build charm4py <version> -j<N> --with-production
```

where `<version>` varies based on the system and communication layer, and `<N>` is the number of processes to use for compiling. For help in choosing the correct `<version>`, please refer to the Charm++ manual and the README in Charm++'s root directory.

After the library has been built, continue with `python3 setup.py install` in the Charm4py source root directory.

CHAPTER 4

Running

Charm4py includes a launcher called `charmrun` to run parallel applications on desktops and small clusters. Supercomputers and some clusters provide their own application launchers (these can also be used to launch Charm4py applications).

4.1 charmrun

After installing Charm4py as explained in the previous section, you can launch applications like this:

```
$ python3 -m charmrun.start +p4 myprogram.py
```

The option `+pN` specifies how many processes to run the application with.

Alternatively, if `charmrun` is in your PATH (this depends on where Charm4py was installed and your system configuration):

```
$ charmrun +p4 myprogram.py
```

You can launch an *interactive shell* using the `++interactive` option, for example:

```
$ python3 -m charmrun.start +p4 ++interactive
```

Charm4py programs accept the same command-line parameters as Charm++.

4.1.1 Running on multiple hosts

`charmrun` can run an application on multiple hosts (e.g. a network of workstations) by passing it a file containing the list of nodes (*nodelist* file). Hosts can be specified by IP address or host name. For example, this is a simple nodelist file specifying four hosts:

```
group mynodes
    host 192.168.0.10
    host 192.168.0.133
    host myhost
    host myhost2
```

The application can be launched like this:

```
$ python3 -m charmrun.start +pN myprogram.py ++nodelist mynodelist.txt
```

With this method, charmrun uses `ssh` to log into remote machines and spawn processes.

charmrun can also use the cluster's `mpiexec` job launcher instead of the built-in ssh method.

See the charmrun manual for more information and alternative ways to work with nodelist files.

Important: If you need to set environment variables on the remote hosts, you can let charmrun do it by adding your export commands to a file called `~/.charmrunrc`.

4.1.2 Using charmrun from a Python program

You can launch a Charm4py application from inside a Python application, and wait for it to complete, in this manner:

```
from charmrun import start

start.start(['+p4', 'myprogram.py'])  # launch parallel application and wait for␣
↪completion
```

Note that you can also use Python's `subprocess` library and launch the same command as you would via the command line.

4.1.3 Troubleshooting

Issue Program hangs with no output when launching with `charmrun`.

Solution This typically occurs when launching the program on multiple hosts, and an error ocurring before starting charm (e.g. syntax error). To diagnose, launch the program on a single host.

4.2 mpirun (or equivalent)

If you have built Charm4py to use MPI, you can launch Charm4py applications using mpirun, mpiexec or other valid method on your system that supports launching MPI applications. For example:

```
$ mpirun -np 4 /usr/bin/python3 myprogram.py
```

See *Install* for instructions on building Charm4py for MPI.

You can launch an *interactive shell* with mpirun like this (or similar command):

```
$ mpirun -np 4 /usr/bin/python3 -m charm4py.interactive
```

Note that the console has limited functionality in terms of tab completion, etc. due to stdin and stdout limitations when using mpirun.

4.3 Using system job launchers

Charm4py applications can also be launched using system job launchers (e.g. aprun, ibrun, SLURM). The exact details of how to do so depend on the system, and typically Charm++ has to be built with a specialized network layer like MPI, GNI or OFI (see Charm++ manual build).

In all cases, the mechanism consists in launching one or multiple Python processes on each node, and passing the main application file to Python. Here is a simple script for SLURM on a Cray-based system:

```
#!/bin/bash -l
#SBATCH -N 8          # number of nodes
#SBATCH -t 00:30:00
#SBATCH -C knl

module load craype-hugepages8M
module load python/3.6-anaconda-4.4

export PYTHONPATH=/path/to/charm4py
PYTHON_EXEC=`which python3`

srun -n 512 -c 1 $PYTHON_EXEC myprogram.py app_param1 app_param2 ...
```

CHAPTER 5

Tutorial

Contents

This is a step-by-step tutorial to introduce the main concepts of Charm4py, and is meant to be done from an interactive session. It is not meant to provide realistic examples, or to cover every possible topic. For examples, you can refer to *Examples*.

To begin, launch an interactive session with 2 processes:

```
$ python3 -m charmrun.start +p2 ++interactive
```

This launches Charm4py with two processes on the local host, with an interactive console running on the first process. In Charm4py, we also refer to processes as Processing Elements (PEs).

5.1 First steps

The interactive console is actually a chare running on PE 0, and the prompt is running inside a coroutine of this chare. Typing:

```
>>> self
<charm4py.interactive.InteractiveConsole object at 0x7f7d9b1290f0>
```

will show that `self` is an InteractiveConsole object. As mentioned, this object exists only on PE 0.

Now, let's look at the `charm` object:

```
>>> charm
<charm4py.charm.Charm object at 0x7f7d9f6d9208>
```

`charm` exists on every PE. It represents the Charm runtime. We can query information from it:

```
>>> charm.myPe()
0
```

Tells us that this process is PE 0.

```
>>> charm.numPes()
2
>>> charm.numHosts()
1
```

The above tells us that we are running Charm4py with 2 PEs on 1 host.

5.2 Chares

In this tutorial, we are going to be defining chares dynamically after the Charm runtime has started, and so these definitions need to be sent to other processes at runtime. Note that non-interactive applications typically have everything defined in the source files (which every process reads at startup).

Let's define a simple chare type. Paste the following in the console:

```
class Simple(Chare):
    def sayHi(self):
        print('Hello from PE', charm.myPe())
        return 'hi done'
```

You will see this:

```
Charm4py> Broadcasted Chare definition
```

We have defined a new chare of type `Simple` and the runtime has automatically broadcasted its definition to other processes. We can now create chares of this type and call their methods:

```
>>> chare = Chare(Simple, onPE=1)  # create a single chare on PE 1
>>> chare
<__main__.SimpleArrayProxy object at 0x7f7d9b129668>
>>> chare.sayHi()
Hello from PE 1
```

It is important to note that `chare` is what is called a *Proxy*. As we can see, remote methods are called via proxies, using regular Python method invocation syntax.

Tip: Proxies are lightweight objects that can be sent to other chares.

The chare we created lives on PE 1, and that is where its method executes. Note that Charm4py automatically collects "prints" and sends them to PE 0, where they are actually printed.

Remote method invocation is asynchronous, returns immediately, and by default does not return anything. We can wait for a call to complete or obtain a return value by requesting a *Future* using `ret=True`:

```
>>> f = chare.sayHi(ret=True)
Hello from PE 1
>>> f
<charm4py.threads.Future object at 0x7f7d9b129f28>
>>> f.get()
'hi done'
```

5.3 Remote method invocation is asynchronous

All method invocations via a proxy are *asynchronous*. Above, we called some remote methods, but they execute so quickly that it is not obvious that it happens asynchronously. To illustrate this more clearly, we will define a method that takes longer to execute.

Paste the following into the console:

```
class AsyncSimple(Chare):
    def sayHi(self):
        time.sleep(5)
        print('Hello from PE', charm.myPe())
        return 'hi done'
```

Now, let's invoke the method:

```
>>> import time
Charm4py> Broadcasted import statement
>>> chare = Chare(AsyncSimple, onPE=1)
>>> chare.sayHi()
```

As we can see, the call returns immediately. We won't see any output until the method completes (after 5 seconds). Now let's see what happens if we want to explicitly wait for the call to complete:

```
>>> f = chare.sayHi(awaitable=True)
>>> f.get()
```

We request a future by making the call `awaitable`. We can then block on the future to wait for completion. **It is important to note that this only blocks the current coroutine** (it does not block the whole process).

Charm also has a nice feature called *quiescence detection* (QD) that can be used to detect when all PEs are idle. We can wait for QD like this:

```
>>> chare.sayHi()
>>> charm.waitQD()
```

5.4 Chare Groups

In many situations we create *collections* of chares, which are distributed across processes by the runtime. First let's look at **Groups**, which are collections with one element per PE:

```
>>> g = Group(AsyncSimple)
>>> g
<__main__.AsyncSimpleGroupProxy object at 0x7f7d9f9f7fd0>
>>> g.sayHi(awaitable=True).get()
Hello from PE 0
Hello from PE 1
```

We created a group of AsyncSimple chares and made an awaitable call. Note that because we don't refer to any specific element, the message is sent to every member (also known as a *broadcast*). We call `get()` on the obtained future, which blocks until the call completes on every member of the group. Note that we didn't get any return values. Let's request return values now:

```
>>> g.sayHi(ret=True).get()
Hello from PE 1
Hello from PE 0
['hi done', 'hi done']
```

As we can see, we got return values from every member. We can refer to specific members by using their index on the proxy. For groups, the index coincides with the PE number:

```
>>> g[1].sayHi(ret=True).get()
'hi done'
Hello from PE 1
```

Chares have one primary collection to which they can belong to, and they have access to the collection proxy via their `thisProxy` attribute. They have access to their index in the collection via the `thisIndex` attribute. For example, define the following chare type:

```
class Test(Chare):
    def start(self):
        print('I am element', self.thisIndex, 'on PE', charm.myPe(),
              'sending a msg to element 1')
        self.thisProxy[1].sayHi()
    def sayHi(self):
        print('Hello from element', self.thisIndex, 'on PE', charm.myPe())
```

Now, we will make element 0 send a message to element 1:

```
>>> g = Group(Test)
>>> g[0].start()
I am element 0 on PE 0 sending a msg to element 1
Hello from element 1 on PE 1
```

You can store a proxy referencing an individual element, for later use:

```
>>> elem = g[0]
>>> elem.sayHi()
Hello from element 0 on PE 0
```

5.5 Chare Arrays

Chare Arrays are a more versatile kind of distributed collection, which can have zero or multiple chares on a PE, and chares can migrate between processes.

Let's create an Array of 4 chares of the previously defined type `Test` and see where the runtime places them:

```
>>> a = Array(Test, 4)
>>> a.sayHi()
Hello from element (2,) on PE 1
Hello from element (3,) on PE 1
Hello from element (0,) on PE 0
Hello from element (1,) on PE 0
```

As we can see, it has created two on each PE.

Array elements have N-dimensional indexes (from 1D to 6D), represented by a tuple. For example, let's create a 2 x 2 array instead:

```
>>> a = Array(Test, (2,2))
>>> a.sayHi()
Hello from element (0, 0) on PE 0
Hello from element (0, 1) on PE 0
Hello from element (1, 0) on PE 1
Hello from element (1, 1) on PE 1
>>> a[(1,0)].sayHi()
Hello from element (1, 0) on PE 1
```

5.6 Charm is a chare too

The `charm` object is a chare too (part of a Group), which means it has methods that can be invoked remotely:

```
>>> charm.thisProxy[1].myPe(ret=True).get()
1
```

Calls the method `myPe()` of `charm` on PE 1, and returns the value.

In interactive mode, Charm also exposes `exec` and `eval` for dynamic remote code execution:

```
>>> charm.thisProxy[1].eval('charm.myPe()', ret=True).get()
1
```

Note that remote exec and eval are only enabled by default in interactive mode. If you want to use them in regular non-interactive mode, you have to set `charm.options.remote_exec` to `True` before the charm runtime is started.

5.7 Broadcasting globals

Suppose we want to broadcast and set globals on some or all processes. With what we know, we could easily implement our own way of doing this. For example, we could create a custom chare Group with a method that receives objects and stores them in the global namespace. However, charm provides a convenient remote method to do this:

```
>>> charm.thisProxy.updateGlobals({'MY_GLOBAL': 1234}, awaitable=True).get()
>>> charm.thisProxy.eval('MY_GLOBAL', ret=True).get()
[1234, 1234]
```

As we can see, there is now a global called `MY_GLOBAL` in the main module's namespace on every PE. We can specify the Python module where we want to set the global variables as a second parameter to `updateGlobals`. If left unspecified, it will use `__main__` (which is the same namespace where InteractiveConsole runs).

5.8 Reductions

Reductions are very useful to aggregate data among members of a collection in a way that is scalable and efficient, and send the results anywhere in the system via a callback. We will illustrate this with a simple example. First define the following chare type:

```
class RedTest(Chare):
    def work(self, data, callback):
        self.reduce(callback, data, Reducer.sum)
    def printResult(self, result):
        print('[' + str(self.thisIndex[0]) + '] Result is', result)
```

Now we will create an Array of 20 of these chares and broadcast some data so that they can perform a "sum" reduction. Normally, each chare would provide its own unique data to a reduction, but in this case we broadcast the value for simplicity. As callback, we will provide a future:

```
>>> a = Array(RedTest, 20)
>>> f = Future()
>>> a.work(1, f)
>>> f.get()
20
```

We manually created a future to receive the result, and passed data (int value 1) and the future via a broadcast call. The chares performed a reduction using the received data, and sent the result to the callback, in this case the future. Because we passed a value of 1, the result equals the number of chares. Note that **reductions happen asynchronously**, and don't block other ongoing tasks in the system.

Note: Reductions are performed in the context of the collection to which the chare belongs to: all objects in that particular collection have to contribute for the reduction to complete.

The other main type of callback used in Charm is a remote method of some chare(s). For example, we can send the result of the reduction to element 7 of the array:

```
>>> a.work(1, a[7].printResult)
[7] Result is 20
```

You can even broadcast the result of the reduction to all elements using `a.printResult` as the callback. Try it and see what happens.

Reductions are useful when data that is distributed among many objects across the system needs to be aggregated in some way, for example to obtain the maximum value in a distributed data set or to concatenate data in some fashion. The aggregation operations that are applied to the data are called **reducers**, and Charm4py includes several built-in reducers, including sum, max, min, product and gather. Users can also define their own reducers (see *Reducers*).

It is common to perform reduction operations on arrays:

```
>>> import numpy
>>> f = Future()
>>> a.work(numpy.array([1,2,3]), f)
>>> f.get()
array([20, 40, 60])
```

You can also do *empty reductions* to know when all the elements in a collection have reached a certain point. Simply provide a callback to the `reduce` call and omit the data and reducer.

5.9 Channels

Channels in Charm4py are streams or pipes between chares (currently only point-to-point). They are useful for writing iterative applications where chares always send/recv to/from the same the set of chares.

Here, we will establish a channel between the InteractiveConsole and another chare. First let's define the chare:

```
class Echo(Chare):
    @coro
    def run(self, remote_proxy):
        ch = Channel(self, remote=remote_proxy)
        while True:
            x = ch.recv()
            ch.send(x)
```

Echo chares will establish a channel with whatever chare is passed to them in the `run` method, and will enter an infinite loop where they wait to receive something from the channel and then send it right back:

```
>>> chare = Chare(Echo, onPE=1)
>>> chare.run(self.thisProxy)
>>> ch = Channel(self, remote=chare)
>>> ch.send('hello')
>>> ch.recv()
'hello'
>>> ch.send(1,2,3)
>>> ch.recv()
(1, 2, 3)
```

Note that on calling `recv()` a coroutine suspends until there is something to receive.

5.10 Pool

Charm4py also has a distributed pool of workers that can be used to execute transient tasks in parallel, where tasks are defined as Python functions. This pool automatically distributes tasks across processes and even multiple hosts.

A common operation is `map`, which applies a function in parallel to the elements of an iterable and returns the list of results. For example:

```
>>> charm.pool.map(abs, range(-1,-20,-1))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

If your tasks are very small, you will want to group them into chunks for efficiency. Pool can do this for you with the `chunksize` parameter (see *Pool*).

Note that the pool of workers reserves PE 0 for a scheduler, so there are P-1 workers (P being the number of PEs). So you might want to adjust the number of processes accordingly.

Tip: Tasks themselves can use the pool to create and wait for other tasks, which is useful for implementing recursive parallel algorithms and state space search (or similar) algorithms. There are examples of this in the source code repository.

CHAPTER 6

Examples

There are several examples in the source code repository, with documentation and comments: https://github.com/UIUC-PPL/charm4py/tree/master/examples

These include:

- A simple distributed job/task scheduler.
- Recursive parallel Fibonacci calculator using *Pool* to spawn tasks. Includes a Numba accelerated version.
- N-Queen problem parallel solver, using a simple state space search algorithm implemented with *Pool* tasks. Includes a Numba accelerated version.
- Jacobi iteration on a 2D array. Can use Numba.
- 2D particle simulation with dynamic load balancing via migratable chares.
- 2D wave simulation displaying a real-time animation. Can use Numba.

CHAPTER 7

Charm

The object `charm4py.charm` exists on every process and represents the Charm4py runtime. It is available after importing `charm4py`. To start the application, you register your *Chare* classes and start the runtime by calling `charm.start()`.

`charm` is also a chare, and some of its methods can be called remotely, via its `thisProxy` attribute.

The API of `charm` is described in this section.

7.1 Start and exit

- **charm.start(entry=None, classes=[], modules=[], interactive=False)**:

  Start the runtime system. This is required on all processes, and registers *Chare* classes with the runtime.

  entry is the user-defined entry point to start the application. The runtime transfers control to this entry point after it has initialized. *entry* can be a Python function or a chare type. If it is a chare type, the runtime will create an instance of this chare *on PE 0*, and transfer control to the chare's constructor. The entry point function (or chare constructor) must have only one parameter, which is used to receive the application arguments.

  If *interactive* is `True`, an entry point is not needed and instead Charm4py will transfer control to an interactive prompt (Read-Eval-Print Loop) on PE 0.

  On calling `charm.start()`, Charm4py automatically registers any chare types that are defined in the `__main__` module. If desired, a list of chare types can also be passed explicitly using the *classes* optional parameter. These must be the classes that are to be registered, and can be classes defined in other modules. The *modules* parameter can be used to automatically register any chare types defined inside the given list of modules. These modules are to be given by their names.

  Note: If `charm4py` is imported and the program exits without calling `charm.start()`, a warning will be printed. This is to remind users in case they forget to start the runtime (otherwise the

program might hang or exit without any output).

- **charm.exit(exit_code=0)**:

  Exits the parallel program, shutting down all processes. Can be called from any chare or process after the runtime has started. The *exit_code* will be received by the OS on exit.

  This method can be called remotely. For this reason, it can also be used a callback (of reductions, etc). For example: `self.reduce(charm.thisProxy.exit)`.

  Note: Calling Python's `exit()` function from a chare has the same effect (Charm4py intercepts the SystemExit exception and calls `charm.exit()`.

- **charm.abort(message)**:

  Aborts the program, printing the specified *message* and a stack trace of the PE which aborted. It can be called from any chare or process after the runtime has started.

7.2 Broadcasting globals

- **charm.updateGlobals(globals_dict, module_name='__main__')**:

  Update the globals dictionary of module *module_name* with the key/value pairs from *globals_dict*, overwriting existing keys.

  This can only be called as a remote method.

  Example:

```
# broadcast global 'X' to all processes, wait for completion
charm.thisProxy.updateGlobals({'X': 333}, awaitable=True).get()
```

7.3 Query processor and host information

- **charm.myPe()**:

  Returns the PE number on which the caller is currently running.

  Note: Some chares can migrate between PEs during execution. As such, the value returned by `myPe()` can vary for these chares.

- **charm.numPes()**:

  Returns the total number of PEs that the application is running on.

- **charm.myHost()**:

  Returns the host number on which the caller is running.

- **charm.numHosts()**:

  Returns the total number of hosts on which the application is running on.

- **charm.getHostPes(host)**:

Return the list of PEs on the specified *host* (given by host number).

- **charm.getHostFirstPe(host)**:

  Return the first PE on the specified *host* (given by host number).

- **charm.getHostNumPes(host)**:

  Return the number of PEs on the specified *host* (given by host number).

- **charm.getPeHost(pe)**:

  Return the host number on which *pe* resides.

- **charm.getPeHostRank(pe)**:

  Returns the local rank number of *pe* on the host on which it resides.

7.4 Waiting for events and completion

You can obtain *Futures* when calling remote methods, to wait for completion (see *Proxies*).

`charm` has the following methods related to waiting for events:

- **charm.awaitCreation(*proxies)**:

  Suspends the current coroutine until all of the chares in the collections referenced by the given proxies have been created on the system (in other words, until their constructors have been called).

  Note: The coroutine must have triggered the creation of the collections.

- **charm.wait(awaitables)**:

  Suspends the current coroutine until the objects in *awaitables* become ready. The objects supported are *Futures* and *Channels*.

- **charm.iwait(awaitables)**:

  Iteratively yield objects from *awaitables* as they become ready. The objects supported are *Futures* and *Channels*. This can only be called from coroutines.

  Warning: Do not suspend the coroutine until `iawait` has finished yielding all the objects.

- **charm.startQD(callback)**

  Start Quiescence Detection (QD). Quiescence is defined as the state in which no PE is executing a remote method, no messages are awaiting processing, and there are no messages in flight. When QD is reached, the runtime will call the *callback*. The callback must be a *Future* or the remote method of a chare(s) (specified by `proxy.method`, where proxy can be any type of proxy, including a proxy to a single element or a whole collection).

- **charm.waitQD()**

  Suspend the current coroutine until Quiescence Detection is reached.

7.5 Timer-based scheduling

- **charm.sleep(secs)**

  If this is called from a coroutine, it suspends the coroutine until at least *secs* seconds have elapsed (the process is free to do other work in that time). If it is not called from a coroutine, it is equivalent to doing `time.sleep(secs)` which puts the process to sleep.

- **charm.scheduleCallableAfter(callable_obj, secs, args=[])**

  Schedule *callable_obj* to be called after *secs* seconds. The callable can be any Python callable, as well as *Futures* and the remote method of a chare(s) (specified by `proxy.method`, where proxy can be any type of proxy, including a proxy to a single element or a whole collection). A list of arguments can be passed via *args* (the callable will be called with these arguments). Note that this method only guarantees that the callable is called after *secs* seconds, but the exact time depends on the work the PE is doing.

7.6 Sections

- **charm.split(proxy, numsections, section_func=None, elems=None)**:

  Split the collection referred to by *proxy* into sections. See *Sections: Split, Slice and Combine Collections* for more information.

- **charm.combine(*proxies)**:

  Combine the collections referenced by *proxies* into one collection, returning a section proxy. See *Sections: Split, Slice and Combine Collections* for more information.

7.7 Remote code execution

Note: These are disabled by default. Set `charm.options.remote_exec` to `True` to enable.

- **charm.exec(code, module_name='__main__')**:

  Calls Python's `exec(code)` on this PE using the specified module as *globals*. *code* is a string containing Python code.

  This can only be called as a remote method.

- **charm.eval(expression, module_name='__main__')**:

  Calls Python's `eval(expression)` on this PE using the specified module as *globals*.

  This can only be called as a remote method.

7.8 Profiling

- **charm.printStats()**:

  Print profiling metrics and statistics. Profiling must have been enabled by setting `charm.options.profiling` to `True` before calling `charm.start()`. See *Profiling* for more information.

This can be called as a remote method.

Example:

```
# print stats of PE 2 and wait for completion
charm.thisProxy[2].printStats(awaitable=True).get()
```

7.9 charm.options

You can set runtime options via the `charm.options` object, which has the following attributes:

- **local_msg_optim** (default=True): if `True`, remote method arguments sent to a chare that is in the same PE as the caller will be passed by reference (instead of copied or serialized). Best performance is obtained when this is enabled.
- **local_msg_buf_size** (default=50): size of the pool used to store "local" messages (see previous option).
- **pickle_protocol** (default=-1): determines the pickle protocol used by Charm4py. A value of `-1` tells `pickle` to use the highest protocol number (recommended). Note that not every type of argument sent to a remote method is pickled (see *Serialization*).
- **profiling** (default=False): if `True`, Charm4py will profile the program and collect timing and message statistics. See *Profiling* for more information. Note that this will affect performance of the application.
- **quiet** (default=False): suppresses the initial Charm++ and Charm4py output.
- **remote_exec** (default=False): if `True`, allows remote calling of `charm.exec()` and `charm.eval()`.

CHAPTER 8

Chares and Proxies

8.1 Chare

An application defines new chare types by subclassing from `charm4py.Chare`. These can have their own custom attributes and methods. In addition, chare classes have to be registered with the runtime when calling `charm.start()` (see *Charm*).

Every chare instance has the following properties:

8.1.1 Attributes

- **thisProxy**:

  If the chare is part of a collection, this is a proxy to the collection to which the element belongs to. Otherwise it is a proxy to the individual chare.

- **thisIndex**:

  Index of the chare in the collection to which it belongs to.

8.1.2 Methods

- **reduce(self, callback, data=None, reducer=None, section=None)**:

  Perform a reduction operation by giving this chare's contribution (see *Reductions*). If *section* is `None`, the reduction is performed across the elements of the primary collection to which the chare belongs to. If *section* is a section proxy, the reduction is performed across the elements of the section (note that the chare must belong to that section).

  The method must be called by all members to perform a successful reduction. *callback* will receive the result of the reduction. It can be a *Future* or the remote method of a chare(s) (specified by `proxy.method`, where proxy can be any type of proxy, including a proxy to a single element or a whole collection). *data* is this chare's contribution to the reduction; *reducer* is the reducer function to apply (see *Reducers*).

It is possible to do "empty" reductions (if no data and reducer are given).

- **allreduce(self, data=None, reducer=None, section=None)**:

  Same as `reduce` but the call will return a *Future* which the caller can use to wait for the result (this means that the result of the reduction is sent to all callers). Can only be called from coroutines.

- **AtSync(self)**:

  Notify the runtime that this chare is ready for load balancing. If load balancing is enabled, load balancing starts on this PE once all of the chares that use "AtSync" have called this. When you create a chare array you specify if its chares use AtSync or not (see *Array*). Load balancing starts globally once all of the PEs have started load balancing.

- **migrate(self, toPe)**:

  Requests migration of the chare to the specified PE. The chare must be *migratable*.

  > **Caution:** This should be called via a proxy so that it goes through the scheduler, for example: `proxy.migrate(toPe)`.
  >
  > Also note that it is unusual for applications to have to manually migrate chares. Instead, applications should delegate to the runtime's load balancing framework.

- **migrated(self)**:

  This is called after a chare has migrated to a new PE. This method is empty, and applications can redefine it in subclasses.

- **setMigratable(self, migratable)**:

  Set whether the chare is migratable or not (*migratable* is a bool). If a chare is not migratable and load balancing is enabled, the load balancing framework will not migrate it. All array chares are migratable by default.

8.1.3 Remote methods

Any user-defined methods of chares can be invoked remotely (via *Proxies*). Note that methods can also be called locally (using standard Python object method invocation). For example, a chare might invoke one of its own methods by doing `self.method(*args)` thus bypassing remote method invocation. Note that in this case the method will be called directly and will not go through the runtime or scheduler.

8.2 Creating single chares

Typically, chares are created as parts of collections (see *Groups* and *Arrays*). You can, however, also create individual chares using the following syntax:

- **Chare(chare_type, args=[], onPE=-1)**:

  where *chare_type* is the type of chare you want to create. *args* is the list of arguments to pass to its constructor. If *onPE* is *-1*, the runtime decides on which PE to create it. Otherwise it will create the chare on the specified PE. This call returns a proxy.

  You can create any number of chares (of the same or different types).

Note: This call is asynchronous: it returns immediately without waiting for the chare to be created. See `charm.awaitCreation()` for one mechanism to wait for creation.

8.3 Proxies

Proxy classes do not exist a priori. They are generated at runtime using metaprogramming, based on the definition of the chare types that are registered when the runtime is started.

Proxy objects are returned when creating chares or collections, and are also stored in the `thisProxy` attribute of chares.

Tip: A proxy object is lightweight and can be sent to any chare(s) in the system via remote methods.

Their methods can also be sent to other chares to use as callbacks (see example below).

Proxies have the same methods as the chare that they reference. Calling those methods will result in the method being invoked on the chare(s) that the proxy references, regardless of the location of the chare.

The syntax to call a remote method is:

proxy.remoteMethod(*args, **kwargs, awaitable=False, ret=False):

> Calls the method of the chare(s) referenced by the proxy. This is a remote method invocation. If the proxy references a collection, a broadcast call is made and the method is invoked on all chares in the collection. Otherwise, the method is called on an individual chare. The call returns immediately and does not wait for the method to be invoked at the remote chare(s).
>
> If *awaitable* is `True`, the call returns a *Future*, which can be used to wait for completion. This also works for broadcast calls (wait for the call to complete on every element).
>
> If *ret* is `True`, the call returns a *Future*, which can be used to wait for the result. This also works for broadcast calls. In this case, the return value will be a list of return values, sorted by element index.
>
> If *ret* or *awaitable* are `True` and the remote method throws an unhandled exception, the exception is propagated to the caller (even if the caller is in another PE). The exception is raised at the caller when it queries the future.

Proxies the refer to collections can be **sliced** to obtain section proxies (see *Sections: Split, Slice and Combine Collections*).

All proxies implement `__eq__` and `__hash__`, with correct results between proxies generated locally and those obtained from a remote PE. This allows, for example, checking proxies for equality, using them as dictionary keys or inserting in sets.

8.3.1 Example

```
from charm4py import charm, Chare, Group

class A(Chare):

    def start(self):
        b_proxy = Chare(B)
        # call work and send one of my methods to use as callback
```

(continues on next page)

(continued from previous page)

```
        b_proxy.work(self.thisProxy.recvResult)

    def recvResult(self, result):
        print('Result is', result)
        exit()

class B(Chare):

    def work(self, callback):
        # ... do work ...
        result = ...
        callback(result)

def main(args):
    a_proxy = Chare(A)
    a_proxy.start()

charm.start(main)
```

CHAPTER 9

Chare Collections

9.1 Group

`charm4py.Group` is a type of collection where there is one chare per PE. These chares are not migratable and are always bound to the PE where they are created. Elements in groups are indexed by integer ID, which for each element coincides with the PE number where it is located.

Groups are created using the following syntax:

- **Group(chare_type, args=[], onPEs=None)**:

  Where *chare_type* is the type of chares that will constitute the group. The list of arguments to pass to the constructor of each element is given by *args*. If *onPEs* is None, creates one instance on every PE and returns a group proxy. If *onPEs* is a list of PEs, only creates instances on those PEs, and returns a section proxy. In this case, the `thisProxy` member of each instance will be a section proxy, and broadcasts/reductions will only involve the specified PEs.

  Any number of groups (of the same or different chare types) can be created.

  Note: The call to create a Group is asynchronous: it returns immediately without waiting for the elements to be created. See `charm.awaitCreation()` for one mechanism to wait for creation.

9.1.1 Group Proxy

A Group proxy references a chare group and its elements. A group proxy is returned when creating a Group (see above) and can also be accessed from the attribute `thisProxy` of the elements of the group. Like any proxy, group proxies can be sent to *any* chares in the system.

Methods

- **self[index]**: return a new proxy object which references the element in the group with the given *index*.

Group proxies can be **sliced** to obtain section proxies (see *Sections: Split, Slice and Combine Collections*).

9.2 Array

`charm4py.Array` is a type of distributed collection where chares have n-dimensional indexes (represented by an integer n-tuple), and members can exist anywhere on the system. As such, there can be zero or multiple elements of a chare array on a given PE, and elements can migrate between PEs.

Arrays are created using the following syntax:

- **Array(chare_type, dims=None, ndims=-1, args=[], map=None, useAtSync=False)**:

  Where *chare_type* is the type of chares that will constitute the array. There are two modes to create an array:

  1. Specifying the bounds. *dims* is an n-tuple indicating the size of each dimension. The number of elements that will be created is the product of the sizes of every dimension. For example, `dims=(2, 3, 5)` will create an array of 30 chares with 3D indexes. If it is a 1D array, *dims* can also be the number of elements in the array.
  2. Empty array of unspecified bounds, when `dims=None`. *ndims* indicates the number of dimensions to be used for indexes.

  The list of arguments to pass to the constructor of each element is given by *args*. *map* can optionally be used to specify an `ArrayMap` for initial mapping of chares to PEs (see below). It must be a proxy to the map. If unspecified, the system will choose a default mapping. If the elements of this array will use AtSync for load balancing *useAtSync* must be `True` (see *Chare*).

  The call to create an array returns a proxy to the array.

  Any number of arrays (of the same or different chare types) can be created.

  Note: The call to create an array returns immediately without waiting for all the elements to be created. See `charm.awaitCreation()` for one mechanism to wait for creation.

Important: Arrays with unspecified bounds support dynamic insertion of elements via the array proxy (see below). Note that these types of arrays can be sparse in the sense that elements need not have contiguous indexes. Elements can be inserted in any order, from any location, at any time.

9.2.1 Array Proxy

An Array proxy references a chare array and its elements. An array proxy is returned when creating an Array (see above) and can also be accessed from the attribute `thisProxy` of the elements of the array. Like any proxy, array proxies can be sent to *any* chares in the system.

Methods

- **self[index]**: return a new proxy object which references the element in the array with the given *index*.
- **self.ckInsert(index, args=[], onPE=-1)**: Insert an element with *index* into the array. This is only valid for arrays that were created empty (with unspecified bounds). *args* is the list of arguments passed to the constructor of the element. *onPE* can be used to indicate on which PE to create the element.

- **self.ckDoneInserting**(): This must be used when finished adding elements with `ckInsert`.

Array proxies can be **sliced** to obtain section proxies (see *Sections: Split, Slice and Combine Collections*).

9.2.2 ArrayMap

An `ArrayMap` is a special type of Group whose function is to customize the initial mapping of chares to PEs for a chare Array.

A custom ArrayMap is defined by writing a new class that inherits from `ArrayMap`, and defining the method `procNum(self, index)`, which receives the index of an array element, and returns the PE number where that element must be created.

To use an ArrayMap, it must first be created like any other Group, and the proxy to the map must be passed to the Array constructor (see above).

Note that array elements may migrate after creation and the ArrayMap only determines the initial placement.

CHAPTER 10

Reductions

A reduction is a distributed and scalable operation that reduces data distributed across chares into a smaller set of data. A reduction involves the chares in a collection (Group, Array or Section). They are started by the elements calling their `Chare.reduce()` or `Chare.allreduce()` methods (see *Chare*).

Important: Reduction calls are asynchronous and return immediately. Chares can start multiple reduction operations at the same time, but every chare in the same collection must contribute to reductions in the same order.

10.1 Reducers

`charm4py.Reducer` contains the reducer functions that have been registered with the runtime. Reducer functions are used in reductions, to aggregate data across the members of a chare collection. `Reducer` has the following built-in attributes (reducers) for use in reductions:

- `max`: max function. When contributions are vectors (lists or arrays) of numbers, the reduction result will be the pairwise or "parallel" maxima of the vectors.
- `min`: min function. Pairwise minima in the case of vector contributions.
- `sum`: sum function. Pairwise sum in the case of vector contributions.
- `product`: product function. Pairwise product in the case of vector contributions.
- `logical_and`: logical and. Requires bool values or arrays of bools.
- `logical_or`: logical or. Requires bool values or arrays of bools.
- `logical_xor`: logical xor. Requires bool values or arrays of bools.
- `gather`: Adds contributions to a Python list, and sorts the list based on the index of the contributors in their collection.

10.1.1 Registering custom reducers

To register a custom reducer function:

- **Reducer.addReducer(func, pre=None, post=None)**:

  Where *func* is a Python function with one parameter (list of contributions), and must return the result of reducing the given contributions. *pre* is optional and is a function intended to pre-process data passed in reduce calls. It must take two parameters `(data, contributor)`, where `data` is the data passed in a reduce call and `contributor` is the chare object. *post* is optional and is a function intended to post-process the data after the whole reduction operation has completed. It takes only one parameter, which is the reduced data.

To refer to a custom reducer:

`Reducer.name`, where `name` is the name of the function that was passed to `addReducer`.

10.2 Example

```
from charm4py import charm, Chare, Group, Reducer
import numpy as np

DATA_SIZE = 100
NUM_ITER = 20

class A(Chare):

    def __init__(self):
        self.data = np.zeros(DATA_SIZE)
        self.iteration = 0

    def work(self):
        # ... do some computation, modifying self.data ...
        # do reduction and send result to element 0
        self.reduce(self.thisProxy[0].collectResult, self.data, Reducer.sum)

    def collectResult(self, result):
        # ... do something with result ...
        self.iteration += 1
        if self.iteration == NUM_ITER:
            exit()
        else:
            # continue doing work
            self.thisProxy.work()

def main(args):
    g = Group(A)
    g.work()

charm.start(main)
```

CHAPTER 11

Futures

Futures are objects that act as placeholders for values which are unknown at the time they are created. Their main use is to allow a coroutine to suspend, waiting until a message or value becomes available without having to exit the coroutine, and without blocking the rest of the coroutines/tasks in its process.

11.1 Creation

Futures can be returned by the runtime when invoking remote methods (see *Proxies*). This allows the caller to continue doing work and wait for the return value at the caller's convenience, or wait for the method to complete.

Futures can also be created explicitly by calling:

- **charm4py.Future(num_vals=1)**:

  Returns a new future object accepting *num_vals* number of values. A future created in this way can be sent to any chare(s) in the system by message passing, with the purpose of allowing remote chares to send values to its origin.

  Note: Futures can only be created from coroutines.

11.2 Methods

- **get(self)**:

  Return the value of the future, or list of values if created with `senders > 1`. The call will block if the value(s) has not yet been received. This can only be called from a coroutine, by the chare that created the future.

  If a future receives an Exception, it will raise it on calling this method.

- **send(self, value=None)**:

Send *value* to the chare waiting on the future. Can be called from any chare.

- **__call__(self, value=None)**:

  This makes futures **callable**, providing a generic callback interface. Calling a future is the same as using the `send()` method.

11.3 Future as callback

Futures are callable (see above) and can also be used as reduction callbacks.

11.4 Example

```
from charm4py import charm, Chare, Array, Future, Reducer

class A(Chare):

    def work(self, future):
        # ...
        result = # ...
        # use future as reduction callback (send reduction result to future)
        self.reduce(future, result, Reducer.sum)

def main(args):
    array_proxy = Array(A, charm.numPes() * 8)
    f = Future()
    array_proxy.work(f)
    result = f.get()  # wait for work to complete on all chares in array
    exit()

charm.start(main)
```

CHAPTER 12

Channels

Channels are streamed connections or pipes between a pair of chares. They simplify the expression of sends/receives from inside a coroutine without having to exit the coroutine.

12.1 Creation

To create a Channel:

- **Channel(chare, remote):**

  Establish a channel between *chare* and *remote*, returning a channel object.

  Note that *chare* is an actual chare object, not a proxy. *remote* is a proxy to the remote chare.

  Channels do not have to be created from coroutines, but they can only be used from coroutines.

  There is no restriction on the number of channels that a chare can establish, and it can establish multiple channels with the same remote.

12.2 Methods

Channel objects have the following methods:

- **send(self, *args):**

  Send the arguments through the channel to the remote chare.

- **recv(self):**

  Receives arguments (unpacked) from the channel. Messages are received in order.

12.3 Example

```
from charm4py import charm, Chare, Array, coro, Channel

NUM_ITER = 100

class A(Chare):

    def __init__(self, numchares):
        myidx = self.thisIndex[0]
        neighbors = []
        neighbors.append(self.thisProxy[(myidx + 1) % numchares])
        neighbors.append(self.thisProxy[(myidx - 1) % numchares])
        self.channels = []
        for nb in neighbors:
            self.channels.append(Channel(self, remote=nb))

    @coro
    def work(self):
        for i in range(NUM_ITER):
            for ch in self.channels:
                ch.send(x, y, z)

            for ch in charm.iwait(self.channels):
                x, y, z = ch.recv()
                # ... do something with data ...

def main(args):
    numchares = charm.numPes() * 8
    array = Array(A, numchares, args=[numchares])
    future = array.work(awaitable=True)
    future.get()
    exit()

charm.start(main)
```

CHAPTER 13

Sections: Split, Slice and Combine Collections

Sections are chare collections formed from subsets of other collections. You can form sections by splitting, slicing and combining other collections (Groups, Arrays, Sections) in arbitrary ways.

13.1 Creating sections

Caution: Creating sections involves messaging. When creating sections using any of the below mechanisms, store and reuse the resulting section proxies and do not repeatedly create the same sections.

- **charm.split(proxy, numsections, section_func=None, elems=None)**

  Split a chare collection into *numsections* number of sections. Returns a list of section proxies.

  The parameter *proxy* is a Group, Array or Section proxy, that refers to the collection to split.

  There are two methods of specifying sections. The first, and preferred method, is to use *section_func*, which is a function that takes a chare object and returns the list of sections to which the object belongs to (sections identified from 0 to numsections-1). It is evaluated at the PE where the object lives (the function must be defined on that PE). If the object is not part of any section, it must return a negative number (or empty list).

  The second method consists in using *elems*, which is a list of lists of chare indexes that are part of each section. For very large sections, this method can be much more expensive than using a *section_func*. Also, it is not recommended for collections where multiple elements have the same index, as it can't discriminate between them (such collections can be obtained using the combine operation -see below-).

  Tip: Elements can be part of multiple sections if desired.

- **Proxy slicing:**

This is a shorthand notation to obtain one section from a proxy, using slicing syntax instead of `split`:

- **proxy[[start]:[stop][:step]]**: for group proxies.
- **proxy[start_0:stop_0[:step_0], ..., start_n-1:stop_n-1[:step_n-1]]**: for n-dimensional array proxies.

See examples below.

- **charm.combine(*proxies)**

  Combines multiple collections into one. Returns a section proxy.

  The parameter *proxies* is a list of proxies that refer to the collections to combine. Collection and chare types don't have to match as long as the methods that will be called via the section proxy have the same signature.

  Note that sending one broadcast to a combined collection is more efficient than sending a separate broadcast to each component. Similarly for reductions.

  Important: The proxy returned by combine can be used for broadcast and reductions on the combined collection, and can also be split. But it cannot be used for sending messages to individual elements in the combined collection.

13.2 Examples

The following example first creates a group of chares and then creates a section from elements on even-numbered PEs:

```
from charm4py import charm, Chare, Group

class Test(Chare):
    def sayHi(self):
        print('Hello from', self.thisIndex)

def sectionNo(obj):
    if obj.thisIndex % 2 == 0:
        return [0]
    else:
        return []

def main(args):
    g = Group(Test)
    # creates one section of elements on even-numbered PEs
    secProxy = charm.split(g, 1, sectionNo)[0]
    # this does the same thing with slicing notation
    secProxy2 = g[::2]
    secProxy.sayHi(awaitable=True).get()
    exit()

charm.start(main)
```

The following example creates a 4 x 4 array of chares, and splits it into 4 sections. It then sends the section proxies to the chares, and tells the first section to perform a section reduction:

```
from charm4py import charm, Chare, Array, Future, Reducer

class Test(Chare):
    def recvSecProxies(self, proxies):
        self.secProxy = proxies[sectionNo(self)]
    def doreduction(self, future):
        self.contribute(1, Reducer.sum, future, self.secProxy)

def sectionNo(obj):
    return obj.thisIndex[0]  # first index determines the section number

def main(args):
    a = Array(Test, (4, 4))  # create a 4 x 4 array
    # split array into 4 sections
    secProxies = charm.split(a, 4, sectionNo)
    a.recvSecProxies(secProxies, awaitable=True).get()  # blocks until proxies␣
↪received
    f = Future()
    # tell section 0 to perform a reduction
    secProxies[0].doreduction(f)
    print(f.get())  # returns 4
    exit()

charm.start(main)
```

This final example creates two 4 x 4 chare arrays, combines them into one section, and broadcasts a message to this section. It then creates 4 sections, each of which spans subsets of both arrays, and broadcasts a message to each section:

```
from charm4py import charm, Chare, Array

class Test(Chare):
    def sayHi(self):
        print('Hello from', self.thisIndex)

def sectionNo(obj):
    return obj.thisIndex[0]  # first index determines the section number

def main(args):
    a1 = Array(Test, (4, 4))  # create a 4 x 4 array
    a2 = Array(Test, (4, 4))  # create a 4 x 4 array
    combined = charm.combine(a1, a2)
    combined.sayHi()  # broadcast to all members of a1 and a2
    # make 4 cross-array sections involving the two arrays
    secProxies = charm.split(combined, 4, sectionNo)
    futures = []
    for proxy in secProxies:
        futures.append(proxy.sayHi(awaitable=True))
    charm.wait(futures)
    exit()

charm.start(main)
```

CHAPTER 14

Pool

Pool is a library on top of Charm4py that can schedule sets of "tasks" among the available hosts and processors. Tasks can also spawn other tasks. A task is simply a Python function. There is only one pool that is shared among all processes. Any tasks, regardless of when or where they are spawned, will use the same pool of distributed workers, thereby avoiding unnecessary costs like process creation or creating more processes than processors.

Warning: `charm.pool` is experimental, the API and performance (especially at large scales) is still subject to change.

Note: The current implementation of `charm.pool` reserves process 0 for a scheduler. This means that if you are running Charm4py with N processes, there will be N-1 pool workers, and thus N-1 is the maximum speedup using the pool. You might want to adjust the number of processes accordingly.

The pool can be used at any point after the application has started, and can be used from any process. Note that there is no limit to the amount of "jobs" that can be sent to the pool at the same time.

The API of `charm.pool` is:

- **map(func, iterable, chunksize=1, ncores=-1)**

  This is a parallel equivalent of the map function, which applies the function *func* to every item of *iterable*, returning the list of results. It divides the iterable into a number of chunks, based on the *chunksize* parameter, and submits them to the pool, each as a separate task. This method blocks the current coroutine until the result arrives.

  The parameter *ncores* limits the job to use a specified number of cores. If this value is negative, the pool will use all available cores (note that the total number of available cores is determined at application launch).

  Use the `@coro` decorator on your functions if you want them to be able to suspend (for example, if they create other tasks and need to wait for the results).

- **map_async(func, iterable, chunksize=1, ncores=-1)**

This is the same as the previous method but immediately returns a *Future*, which can be queried asynchronously.

- **Task(func, args, ret=False, awaitable=False)**

  Create a single task to run the function *func*. The function will receive *args* as unpacked arguments.

  By default this returns nothing. If *awaitable* is `True`, the call returns a *Future*, which can be used to wait for completion of the task. If *ret* is `True`, the call returns a *Future*, which can be used to wait for the task's return value.

  Creating a single task is similar to using `map_async(func, iterable)` with an iterable of length one. There are, however, some subtle differences:

  – By default it doesn't create a future or receive a result, which is less expensive.

  – The task can spawn other tasks without having to be a coroutine (if it doesn't request a future).

  – The task receives the arguments unpacked.

14.1 Examples

```
from charm4py import charm

def square(x):
    return x**2

def main(args):
    result = charm.pool.map(square, range(10), chunksize=2)
    print(result)  # prints [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
    exit()

charm.start(main)
```

Note that due to communication and other overheads, grouping items into chunks (with *chunksize*) is necessary for best efficiency when the duration of tasks is very small (e.g. less than one millisecond). How small a task size (aka grain size) the pool can efficiently support depends on the actual overhead, which depends on communication performance (network speed, communication layer used -TCP, MPI, etc-, number of hosts...). The chunksize parameter can be used to automatically increase the grainsize.

```
from charm4py import charm, coro

# Recursive Parallel Fibonacci

@coro
def fib(n):
    if n < 2:
        return n
    return sum(charm.pool.map(fib, [n-1, n-2]))

def main(args):
    print('fibonacci(13)=', fib(13))
    exit()

charm.start(main)
```

CHAPTER 15

Rules and Semantics

15.1 Remote method invocation

Every method invocation *via a proxy* is asynchronous. The call returns immediately, and a message is sent to the destination(s). Therefore, the method is not executed immediately, not even if the destination is in the same process.

This is what happens to arguments passed via remote invocation:

1. If a call is point-to-point and the source and destination are in the same process, arguments are passed by reference in usual Python fashion. In other words, the objects become shared (no serialization or copying is involved, thus making the call efficient). The same is true for destinations of a section broadcast that are in the same process as the source.
2. In all other cases, arguments are serialized and copied into a message (see *Serialization* for more information about serialization).

When a remote method is invoked, the recipient might not be the sole owner of the arguments. This can happen if:

- The destination of a point-to-point send is in the same process as the sender (see case 1 above).
- Is the recipient of a broadcast message and there are multiple recipients on the same process. In this case, all of them receive references to the same Python objects.

The runtime adopts this behavior to guarantee best performance for all applications. But with the above in mind, if an application needs chares to maintain separate copies of specific arguments, it can simply copy them before sending or upon reception as appropiate and required by the application.

15.2 Method execution

The methods of a chare execute on the process where the chare currently lives. When a method starts executing, it runs until either the method completes or suspends (this last case only if it is a coroutine). Note that coroutines are non-preemtive and only yield when they are waiting for something (typically occurs when they are blocked on a future or channel). When a method stops executing, control returns to the scheduler which will start or resume other methods

on that process. Note that if a couroutine C of chare A suspends, the scheduler is free to execute other methods of chare A, even other instances of coroutine C.

CHAPTER 16

Performance Tips

Charm4py will help you parallelize and scale your applications, but it won't make the sequential parts of your code faster. For this, there are several technologies that accelerate Python code, like NumPy, Numba, Cython and PyPy. These are outside the scope of this section, but we highly recommended using Numba. We have found that using Charm4py + Numba, it is possible to build parallel applications entirely in Python that have the same or similar performance as the equivalent C++ application. Many examples in our source code repository use Numba.

This section contains tips to help maximize the performance of your applications by reducing runtime overhead. Overhead becomes apparent at very low method or task granularity and high communication frequency. Therefore, whether these tips actually help depends on the nature of your application and the impact overhead has on it. Also keep in mind that there are other factors besides overhead that can affect performance, and are outside the scope of this section.

Note: Method granularity refers to the time for a chare's remote method to run or, in the case of coroutines, the time the method runs before it suspends and control is switched to a different task.

- For best inter-process communication *on the same host*, an efficient network layer is highly recommended. For example, OpenMPI uses shared memory for inter-process communication and is much faster than Charm++'s TCP communication layer. On supercomputers, you should build Charm++ choosing a network layer that is optimized for the system interconnect. The Charm4py version distributed via pip uses TCP. You have to build Charm++ to use a different network layer (see *Install*).
- If you are sending large arrays of data, use Numpy arrays (or arrays from Python's `array` package) and send each as a separate parameter. This allows Charm4py to directly copy the contents of the arrays to a message that is sent through the network (thus bypassing pickling/serialization libraries). For example: `proxy.method(array1, array2, array3)`.

  In the case of updateGlobals, have each array be an element of the dict, for example: `charm.thisProxy.updateGlobals({'array1': array1, 'array2': array2, ...})`

  With channels, do the following: `ch.send(array1, array2, ...)`

  Note that these types of arguments can be freely intermixed with others not supporting the buffer protocol.

- If you are frequently indexing a proxy (for example `myproxy[3]`) it is more efficient to store the proxy to the individual element and reuse it, for example:

```
elem_proxy = myproxy[3]
for _ in range(100):
  elem_proxy.work(...)
```

- When calling remote methods, it is generally more efficient to use unnamed arguments.
- Avoiding `awaitable=True` and `ret=True` in the critical path can reduce overhead in some cases. Internally, awaitable calls require creating a future and sending it as part of your remote method call. It should always be possible to rewrite code so that notification of completion or results are sent via a separate and explicit method invocation, although this can tend to result in less readable code.
- Make sure profiling is disabled (it is disabled by default). Charm4py prints a warning at startup if it is enabled.
- Charm4py can access the Charm++ shared library using three different technologies: ctypes, cffi and cython. If you are using CPython (the most common implemention of Python), make sure you are using the Cython layer (this is what the pip version of Charm4py uses). If you are using PyPy, make sure you are using the CFFI layer. Charm4py will warn at startup if you are not using the most efficient layer.

CHAPTER 17

Profiling

Charm4py provides basic functionality for profiling. Please note that it may not be suitable for full-blown parallel performance analysis. Essentially, it measures the time spent inside the Charm4py runtime and application, and presents the information broken down by remote methods, and send and receive overhead.

Warning: The profiling tool:

- Does not track time inside the Charm++ scheduler, or time waiting for messages to arrive from the network.
- The tool provides separate timings for each PE. Timings for one PE can be very different from those of another PE (depending on the application), so you should print the information for all the PEs you are interested in.
- Currently the tool does not provide timings for a specific time period, so it can be hard to determine how much work each PE is doing in a specific time interval.

It is important to note that the total application time could be larger than the total time reported for one PE by the profiling tool. This can happen, for example, if the parallel efficiency is not good (e.g. idle processes waiting for work to complete in other processes, load imbalance, ...), or processes spending a lot of time waiting for communication to complete (which depends on the size and number of messages per second, network throughput and latency, efficiency of the communication layer, etc.). If you are seeing significant idle time, one thing that can increase performance is using an efficient communication layer such as MPI (see *Performance Tips*).

In the future, we plan to support Projections which is a full-fledged parallel performance analysis tool for Charm++.

17.1 Usage

Set `charm.options.profiling` to `True` before the runtime starts in order to activate profiling.

To print profiling results at any time, call `charm.printStats()`. This prints timings and message statistics *for the PE where it is called*. A good place to use this is right before exiting the application. You can also invoke this remotely, doing `charm.thisProxy[pe].printStats()` (like any other remote method, you can also wait for its completion).

Here is some example output of the profiler from `examples/particle/particle.py` executed on 4 PEs with `python3 -m charmrun.start +p4 particle.py +balancer GreedyRefineLB`:

```
Timings for PE 0 :
|                              | em    | send  | recv  | total |
------ <class '__main__.Cell'> as Array ------
| __init__                     | 0.001 | 0.001 | 0.0   | 0.002 |
| migrated                     | 0.0   | 0.0   | 0.027 | 0.028 |
| AtSync                       | 0.0   | 0.0   | 0.0   | 0.0   |
| _channelConnect__            | 0.001 | 0.0   | 0.001 | 0.002 |
| _channelRecv__               | 0.037 | 0.0   | 0.056 | 0.093 |
| _coll_future_deposit_result  | 0.0   | 0.0   | 0.0   | 0.0   |
| _getSectionLocations_        | 0.0   | 0.0   | 0.0   | 0.0   |
| getInitialNumParticles       | 0.0   | 0.0   | 0.0   | 0.0   |
| getNbIndexes                 | 0.0   | 0.0   | 0.0   | 0.0   |
| getNumParticles              | 0.0   | 0.0   | 0.0   | 0.0   |
| migrate                      | 0.0   | 0.0   | 0.0   | 0.0   |
| reportMax                    | 0.0   | 0.0   | 0.0   | 0.0   |
| resumeFromSync               | 0.0   | 0.0   | 0.0   | 0.001 |
| run                          | 3.247 | 0.056 | 0.001 | 3.304 |
| setMigratable                | 0.0   | 0.0   | 0.0   | 0.0   |
-------------------------------------------------------------
|                              | 3.288 | 0.058 | 0.086 | 3.432 |
-------------------------------------------------------------
| reductions                   |       |       | 0.0   | 0.0   |
| custom reductions            |       |       | 0.0   | 0.0   |
| migrating out                |       |       | 0.002 | 0.002 |
-------------------------------------------------------------
|                              | 3.288 | 0.058 | 0.088 | 3.434 |

Messages sent: 6957
Message size in bytes (min / mean / max): 0 / 234.319 / 4083
Total bytes = 1.555 MB

Messages received: 6925
Message size in bytes (min / mean / max): 6 / 291.771 / 315390
Total bytes = 1.927 MB
```

The output first shows timings (in seconds) for each chare *type* that was active on the PE. Each row corresponds to a different entry method (i.e. a remote method invoked via a proxy).

Important: Only remote method invocations are measured (not regular function calls). So, if a remote method calls other functions and methods locally (without using a proxy), the time to execute those will be added to its own time.

Timings are shown in four colums:

em Time at the application level (outside of runtime) executing an entry method.

send Time in send calls (like proxy calls, reduce, channel send, etc.)

recv Time between Charm4py layer receiving a message for delivery and the target entry method being invoked.

total Sum of the previous three columns.

The last rows show miscellaneous overheads pertaining to reductions and migration. The last part of the output shows message statistics for remote method invocations (number of messages sent and received and their sizes).

In this example we can see that most of the time is spent inside the "run" method of each Cell. Most of the send

overhead is from the "run" method (it calls channel.send() repeatedly) and most of the receive overhead is in the Cell's `_channelRecv__` method (which is a method of the parent class `Chare` that is called when one of its channels receives a message). There is also some receive overhead due to chares migrating into this PE (method `migrated`).

CHAPTER 18

Serialization

Usually a remote method invocation results in serialization of the arguments into a message that is sent to a remote process. For many situations, Charm4py relies on Python's `pickle` module.

Important: Pickling is bypassed for certain data types that implement the buffer protocol (byte arrays, array.array and NumPy arrays) and is encouraged for best performance. For these, the data is directly copied from its memory buffer in Python into a message in the Charm C++ library for sending. The *Performance Tips* section explains how to take advantage of this.

Pickling can account for much of the overhead of the Charm4py runtime. Fastest pickling is obtained with the C implementation of the `pickle` module (only available in CPython). A general guideline to achieve good pickle performance is to avoid passing custom types as arguments to remote methods in *the application's critical path*. Examples of recommended types to use for best performance include: Python containers (lists, dicts, set), basic datatypes (int, float, str, bytes) or combinations of the above (e.g.: dict containing lists of ints). Custom objects are automatically pickled but can significantly affect the performance of pickle and therefore their use inside the critical path is not recommended.

CHAPTER 19

Contact

You can contact us in the forum for discussion of any topics related to Charm4py or Charm++, or in our mailing list at <charm@cs.illinois.edu>

Please use the GitHub page to report and track issues (https://github.com/UIUC-PPL/charm4py)

CHAPTER 20

Release Notes

This describes the most significant changes. For more detail, see the commit log in the source code repository.

20.1 What's new in v1.0

This is a major release, containing API changes, many new features, bug fixes and performance improvements. The highlights are:

- Many improvements that simplify how Charm4py programs are written. There have been API changes, but most of the old API is preserved for backward compatibility. Please consult the documentation if you encounter any issues, and consider migrating to the new API.
- **Coroutines**: threaded entry methods are now referred to as coroutines (which is a more common term) and the appropiate way to declare them is with the new `@coro` decorator.
  - Significant performance improvement of coroutines, now implemented using the greenlet package.
- **Channels**: can establish channels between arbitrary pairs of chares, and use them to send/receive data inside coroutines.
- **Sections**: it is now possible to split, slice and combine chare collections in arbitrary ways, to form new collections called *sections*, containing some subset or combination of elements. Sections are referenced by section proxies, and the usual operations of broadcast and reductions are supported (uses efficient multicast trees that only involve the PEs where the section elements are located).
- Can now create Groups that are constrained to certain PEs, for example: `g = Group(MyChare, onPEs=[0,4,8...])`. This uses the sections implementation underneath.
- Keyword arguments are now correctly supported when calling remote methods.
- Added `charm.wait(awaitables)` that waits for the given objects to become ready (works for futures and channels).
- Added `charm.iwait(awaitables)` that iteratively yields objects as they become ready (works for futures and channels).

- Huge rewrite and revision of examples to use new features (like coroutines and channels), and added documentation and comments. New examples added.
- Same applies to the tutorial and large parts of the documentation.
- Added ARM support (tested on Raspberry Pi 3 B+).
- **Pool**: added an interface to create single tasks. Tasks themselves can spawn any number of other tasks. Python functions decorated with `@coro` can be used as tasks that can wait for messages, other tasks to complete, etc.
- Added Quiescence Detection (QD) support. Please see the manual and tutorial for more information.
- Can now use Futures and proxy methods as *callbacks*, that is: *callable* objects that can be sent to other chares, and which those chares can call to send values back. For example: `someproxy[3].somemethod` is a valid callback that can be sent (requires Python 3+).
- The `ret=True` keyword argument now has the same semantics for broadcast calls as for point-to-point calls: the future will receive the return values. For the broadcast case, it will be a list of return values (of all the called elements), sorted by element index.
- Added `awaitable` keyword argument to proxy calls, to wait for completion without sending return values to the caller.
- Better method to update globals: `charm.updateGlobals(dict, module_name)` is a remote method that can be called to update global values on any PE at any time (typically used as a broadcast call). It has the same rules and semantics as any other proxy call, so it can be waited upon.
- Distributed exception handling: if a future is requested when invoking remote methods (using `awaitable=True` or `ret=True`) and an exception happens in the remote method, it is propagated to the caller.
- **Interactive mode**: several QOL improvements. Note that we consider this mode to still be in beta, and encourage feedback. Some highlights:
  - Improved launch process for interactive mode. Now also works with charmrun in ssh mode (which can be used to launch an interactive session using multiple hosts).
  - Can automatically broadcast import statements to other PEs (on by default).
  - Can automatically broadcast and register Chare definitions (on by default).
  - Exceptions (whether local or remote) should not crash the interactive session anymore (thanks to distributed exception handling mentioned above).
- Chares now have a method called `migrated` that applications can redefine to get notified when a chare has migrated.
- Added `Chare.setMigratable(bool)` to indicate whether a chare that is part of an Array can be migrated or not.
- All proxies implement `__eq__` and `__hash__`, with correct results between proxies generated locally and those obtained from a remote PE. This allows, for example, checking proxies for equality, using them as dictionary keys or inserting in sets.

20.2 What's new in v0.12.3

- Fixed some bugs in charm.pool, one of which could cause charm.pool to hang when tasks return None.

20.3 What's new in v0.12.2

- Fixed a bug which decreased the performance of charm.pool, and could result in high memory usage in some circumstances.
- Added API to obtain information on nodes, for example: the number of nodes on which the application is running, or the first PE of each node.
- Expanded topology-aware tree API to allow obtaining the subtrees of a given PE in a topology-based spanning tree.

20.4 What's new in v0.12

- Added experimental charm.pool library which is similar to Python's multiprocessing Pool, but also works in a distributed setting (multiple hosts), tasks can create other tasks all of which use the same shared pool, and can benefit from Charm++'s support for efficient communication layers such as MPI. See documentation for more information.
- Improved support for building and running with Charm++'s MPI communication layer. See Install and Running sections of the documentation for more information.
- Substantially improved the performance of threaded entry methods by allowing thread reuse.
- Blocking allreduce and barrier is now supported inside threaded entry methods: `result = charm.allReduce(data, reducer, self)` and `charm.barrier(self)`.
- Can now indicate if array elements use AtSync at array creation time by passing `useAtSync=True` in Array creation method.
- Minor bugfixes and improvements.

20.5 What's new in v0.11

- Changed the name of the project from CharmPy to *charm4py* (more information on why we changed the name is in the forum).
- Not directly related to this release, but there is a new forum for charm4py discussions (see contact details). Feel free to visit the forum for discussions, reports, provide feedback, request features and to follow development.
- Support for interactive charm4py shell using multiple processes on one host has been added as a *beta* feature. Please provide feedback and suggestions in the forum or GitHub.
- Uses the recent major release of Charm++ (6.9)
- C-extension module can be built on Windows. Windows binary wheels on PyPI come with the compiled extension module.
- API change: method `Chare.gather()` has been removed to make the name available for user-defined remote methods. Use `self.contribute(data, Reducer.gather, ...)` instead.
- Some methods of `charm` are now remotely callable, like `charm.exit()`. They can be used as any other remote method including as targets of reductions. For example: `self.contribute(None, None, charm.thisProxy[0].exit)`
- Can now use Python exit function instead of `charm.exit()`
- Other small fixes and improvements.

20.6 What's new in v0.10.1

This is a bugfix and documentation release:

- Added core API to docs, and more details regarding installation and running
- Fixed reduction to Future failing when contributing numeric arrays
- Charm4py now requires Charm++ version >= `6.8.2-890` which, among other things, includes fixes for the following Windows issues:
  - Running an application without `charmrun` on Windows would crash
  - Abort messages were sometimes not displayed on exit. On Charm4py, this had the effect that Python runtime errors were sometimes not shown.
  - If running with charmrun, any output prior to charm.start() would not be shown. On Charm4py, this had the effect that Python syntax errors were not shown.

20.7 What's new in v0.10

Installation and Setup

- Charm4py can be installed with pip (`pip install charm4py`) on regular Linux, macOS and Windows systems
- Support setuptools to build, install, and package Charm4py
- Installation from source is much simpler (see documentation)
- charm4py builds include the charm++ library and are relocatable. `LD_LIBRARY_PATH` or similar schemes are no longer needed.
- charm4py does not need a configuration file anymore (it will automatically select the best available interface layer at runtime).

API Changes

- Start API is now `charm.start(entry)`, where `entry` can be a regular Python function, or any chare type. Special Mainchare class is no longer needed.

Performance

- Added Cython-based C-extension module to considerably speed up the interface with the Charm++ library and critical parts of charm4py (currently only with Python 3+).
- Several minor performance improvements

Features

- *Threaded entry methods*: entry methods can run in their own thread when tagged with the `@threaded` decorator. This enables direct style programming with asynchronous remote method execution (also see Futures):
  - The entry point (main function or chare) is automatically threaded by default
  - Added `charm.awaitCreation(*proxies)` to wait for Group and Array creation within the threaded entry method that created them
  - Added `self.wait('condition')` construct to suspend entry method execution until a condition is met
- *Futures*

  - Remote method invocations can optionally return futures with the `ret` keyword: `future = proxy.method(ret=True)`. Also works for broadcasts.
  - A future can be queried to obtain the value with `future.get()`. This will block if the value has not yet been received.
  - Futures can be explicitly created using `future = charm.createFuture()`, and passed to other chares. Chares can send values to the future by calling `future.send(value)`
  - Futures can be used as reduction targets
- Simplified `@when` decorator syntax and enhanced to support general conditions involving a chare's state and remote method arguments. New syntax is `@when('condition')`.
- Can now pass arguments to chare constructors
- Can create singleton chares. Syntax is `proxy = Chare(MyChare, pe)`
- ArrayMap: to customize initial mapping of chares to cores
- Warn if user forgot to call `charm.start()` when launching charm4py programs
- Exposed `migrateMe(toPe)` method of chares to manually migrate a chare to indicated PE
- Exposed LBTurnInstrumentOn/Off from Charm++ to charm4py applications
- Interface to construct topology-aware trees of nodes/PEs

Bug Fixes

- Fixed issues related to migration of chares

Documentation

- Updated documentation and tutorial to reflect changes in installation, setup, addition of Futures and API changes
- Added leanmd results to benchmarks section

Examples and Tests

- Improved performance of `stencil3d_numba.py`, and added better benchmarking support
- Added parallel map example (`examples/parallel-map/parmap.py`)
- Improved output and scaling of several tests when launched with many (> 100) PEs
- Cleaned, updated, simplified several tests and examples by using futures

Profiling

- Fixed issues which resulted in inaccurate timings in some circumstances
- Profiling of chare constructors (including main chare and chares that are migrating in) is now supported

Code

- Code has been structured as a Python package
- Heavy code refactoring. Code simplification in several places
- Several improvements towards PEP 8 compliance of core charm4py code. Indentation of code in `charm4py` package is PEP 8 compliant.
- Improvements to test infrastructure and added Travis CI script

20.8 What's new in v0.9

General

- Charm4py is compatible with Python 3 (Python 3 is the recommended option)
- Added documentation (http://charm4py.readthedocs.io)

API Changes

- New API to create chares and collections: all chare types are defined by inheriting from Chare. To create a group: `group_proxy = Group(MyChare)`. To create an array: `array_proxy = Array(MyChare, ...)`.
- Simplified program start API with automatic registration of chares

Performance

- Bypass pickling of common array types (most notably numpy arrays) by directly copying contents of their buffer into messages. This can result in substantial performance improvement.
- Added optional CFFI-based layer to access Charm++ library, that is faster than existing ctypes layer.
- The `LOCAL_MSG_OPTIM` option (True by default) avoids copying and serializing messages that are directed to an object in the same process. Works for all chare types.

Features

- Support reductions over chare arrays/groups, including defining custom reducers. Numpy arrays and numbers can be passed as data and will be efficiently reduced. Added "gather" reducer.
- Support dynamic insertion into chare arrays
- Allow using int as index of 1D chare array
- `element_proxy = proxy[index]` syntax now returns a new independent proxy object to an individual element
- Added `@when('attrib_name')` decorator to entry methods so that they are invoked only when the first argument matches the value of the specified chare's attribute
- Added methods `charm.myPe()`, `charm.numPes()`, `charm.exit()` and `charm.abort()` as alternatives to CkMyPe, CkNumPes, CkExit and CkAbort

Other

- Improved profiling output. Profiling is disabled by default.
- Improved general error handling and output. Errors in charm4py runtime raise `Charm4PyError` exception.
- Code Examples:
  - Updated stencil3d examples to use the `@when` construct
  - Added particle example (uses the `@when` construct)
  - Add total iterations as program parameter for wave2d
- Added `auto_test.py` script to test charm4py