# charm4py Documentation

***Release 0.12.3***

**Juan Galvez**

**Apr 10, 2019**

# Getting Started

charm4py is a high-level parallel and distributed programming framework with a simple and powerful API, based on migratable Python objects and remote method invocation; built on top of an adaptive C/C++ runtime system providing *speed*, *scalability* and *dynamic load balancing*.

charm4py allows writing parallel and distributed applications in Python based on the Charm++ programming model. Charm++ has seen extensive use in the scientific and high performance computing (HPC) communities across a wide variety of computing disciplines, and has been used to produce several large parallel applications that run on the largest supercomputers, like NAMD.

With charm4py, all the application code can be written in Python. The core Charm++ runtime is implemented in a C/C++ shared library which the `charm4py` module interfaces with.

As with any Python program, there are several methods available to support high-performance functions where needed. These include, among others: NumPy; writing the desired functions in Python and JIT compiling to native machine instructions using Numba; or accessing C or Fortran code using f2py. Another option for increased speed is to run the program using a fast Python implementation (e.g. PyPy).

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 (whether based on Charm++ or MPI), and that scale to hundreds of thousands of cores.

Example applications are in the `examples` subdirectory of the source code repository.

CHAPTER 1

# Features

- **Speed**: Runtime overhead is very low, particularly compared to Python libraries with similar features. Charm4py runs on top of Charm++, a C/C++ runtime system designed to run High-performance computing (HPC) applications and to scale to hundreds of thousands of cores
- Simple and powerful API
- Programming model based on objects, remote method invocation and futures offers easy translation path from serial Python programs
- Asynchronous method invocation and message passing
- Automatic overlap of communication and computation
- Dynamic load balancing
- High-performance communication supported on many systems (RDMA, CMA, ...)

# CHAPTER 2

# Overview

*Chares* are distributed Python objects that live and perform work on a processor[1]. They can be migrated to different processors by the runtime to, for example, dynamically balance load. Many objects of the same or different types can live on one processor. Chares communicate and coordinate between themselves by invoking their methods, which are defined and called as regular Python methods. This works regardless of the location of the caller and callee, and the runtime automatically takes care of location management and of using the most efficient technique for method invocation. For example, if the communicating objects exist on different hosts, Charm will automatically pack the method arguments into a message and send the message to the destination object by the most efficient way possible.[2]

It is generally desirable to have many objects per core so that the runtime has freedom to dynamically balance load and to overlap communication and computation, thus increasing resource utilization and minimizing idle times[3].

[1] We also refer to processors as cores.

[2] Charm4py supports reference passing for objects in the same process, Cross Memory Attach for objects in the same host, RDMA for network communications, collectives optimized for network topology, among other performance features.

[3] The Charm runtime automatically schedules messages, work and dynamically balances load by migrating objects between cores.

CHAPTER 3

# Install

charm4py runs on Linux, macOS, Windows, 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. 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:

```
$ pip install charm4py
```

**Note:** This option selects 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: install numpy and cython (`pip install 'numpy>=1.10.0' cython`)
- PyPy: none

To build the latest *stable* release, do:

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

Or download the source distribution from PyPI, uncompress and run `python 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
$ python 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, Intel OFI and IBM PAMI. 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 `python 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 `python 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:

```
$ python -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:

```
$ python -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:

```
$ charmrun +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.

### 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.

## 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 tutorial assumes that you have installed Charm4py as described in *Install*. You can run any of these examples in an interactive Python shell (using multiple processes) by launching Charm4py in the following manner:

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

and inserting code at the prompt. Note that in interactive mode the runtime is already started when the interactive shell appears, so `charm.start()` does *not* need to be called. For the examples below, you can directly call the main function or, alternatively, just run the body of the main function in the top-level shell.

## 5.1 Program start and exit

To start a Charm program, you need to invoke the `charm.start(entry)` method. We will begin with a simple example:

```
# start.py
from charm4py import charm

def main(args):
```

(continues on next page)

(continued from previous page)

```
    print("Charm program started on processor", charm.myPe())
    print("Running on", charm.numPes(), "processors")
    exit()

charm.start(main)  # call main([]) in interactive mode
```

We need to define an entry point to the Charm4py program, which we refer to as the Charm *main* function. In our example, it is the function called `main` . The main function runs on only one processor, typically processor 0, and is in charge of starting the creation and distribution of work across the system. The main function must take one argument to get the list of command-line arguments. In this example, we are specifying the function `main` as the main function by passing it to the `start` method.

The method `numPes` returns the number of processors (aka Processing Elements) on which the distributed program is running. The method `myPe` returns the processor number on which the caller resides.

An explicit call to `exit()` is necessary to finish the parallel program, shutting down all processes. It can be called from any chare on any processor.

To launch the example with charmrun using 4 processes:

```
$ python -m charmrun.start +p4 start.py
```

## 5.2 Defining Chares

Chares are distributed objects that make up the parallel application (see *Overview*). To define a Chare, simply define a class that is a subclass of `Chare`.

```
from charm4py import Chare

class MyChare(Chare):

    def __init__(self):
        # chare initialization code here

    def work(self, data):
        # ... do something ...
```

Any methods of `MyChare` will be remotely callable by other chares.

For easy management of distributed objects, you can organize chares into distributed collections:

```
# chares.py
from charm4py import charm, Chare, Group, Array

class MyChare(Chare):
    def __init__(self):
        print("Hello from MyChare instance in processor", charm.myPe())

    def work(self, data):
      pass

def main(args):

    # create one instance of MyChare on every processor
    my_group = Group(MyChare)
```

(continues on next page)

(continued from previous page)

```
    # create 3 instances of MyChare, distributed among the cores by the runtime
    my_array = Array(MyChare, 3)

    # create 2 x 2 instances of MyChare, indexed using 2D index and distributed
    # among all cores by the runtime
    my_2d_array = Array(MyChare, (2, 2))

    charm.awaitCreation(my_group, my_array, my_2d_array)
    exit()

charm.start(main)  # call main([]) in interactive mode
```

The above program will create P + 3 + 2*2 chares and print a message for each created chare, where P is the number of processors used to launch the program. This is the output for 2 PEs:

```
$ python -m charmrun.start +p2 chares.py ++quiet
Hello from MyChare instance in processor 0
Hello from MyChare instance in processor 0
Hello from MyChare instance in processor 0
Hello from MyChare instance in processor 0
Hello from MyChare instance in processor 0
Hello from MyChare instance in processor 1
Hello from MyChare instance in processor 1
Hello from MyChare instance in processor 1
Hello from MyChare instance in processor 1
```

It is important to note that creation of chares across the system happens asynchronously. In other words, when the above calls to create collections return, the chares have not yet been created on all PEs. The `awaitCreation` method is used to wait for all the chares in the specified collections to be created.

---

**Note:** Chares can be created at any point once the Charm *main* function has been reached.

---

If a program defines new Chare types in files other than the one used to launch the application, the user needs to pass the names of those modules when starting charm. For example:

```
charm.start(main, ['module1', 'module2'])
```

## 5.3 Remote method invocation

To invoke methods on chares, a remote reference or *proxy* is needed. A proxy has the same methods as the chare that it references. For example, assuming we have a proxy to a `MyChare` object, we can call method `work` like this:

```
# invoke method 'work' on the chare, passing list [1,2,3] as argument
proxy.work([1,2,3])
```

Any number and type of arguments can be used, and the runtime will take care of sending the arguments if the destination is on a different host. We will also refer to invoking a remote method as sending a message.

**Warning:** Make sure that the caller does not modify any objects passed as arguments after making the call. It also should not attempt to reuse them if the callee is expected to modify them. The caller can safely discard any references to these objects if desired.

References to collections serve as proxies to their elements. For example, `my_group` above is a proxy to the group and its elements. To invoke a method on all elements in the group do:

```
my_group.work(x)     # 'work' is called on every element
```

To invoke a method on a particular element do:

```
my_group[3].work(x)  # call 'work' on element with index 3
```

To store a proxy referencing an individual element for later use:

```
elem_3_proxy = my_group[3]
elem_3_proxy.work(x)    # call 'work' on element with index 3 in my_group
```

The above also applies to Chare Arrays. In the case of N-dimensional array indexes:

```
my_array[10,10].work(x)      # call 'work' on element (10,10)
```

**Tip:** Proxies can be sent to other chares as arguments of methods.

For performance reasons, method invocation is always *asynchronous* in Charm4py, i.e. methods return immediately without waiting for the actual method to be invoked on the remote object, and therefore without returning any result. Asynchronous method invocation is desirable because it leads to better overlap of computation and communication, and better resource utilization (which translates to more speed). Note that this does not mean that we cannot obtain a result from a remote chare as a result of calling one of its methods. There are two ways of doing this:

*1. Using Futures:*

The user can request to obtain a future as a result of calling a remote method, by using the keyword `ret`:

```
def work(self):
    # call method 'apply' of chares with index (10,10) and (20,20), requesting futures
    future1 = my_array[10,10].apply(3, ret=True)
    future2 = my_array[20,20].apply(3, ret=True)

    # ... do more work ...

    # I need the results now, call 'get' to obtain them. Will block until they arrive,
    # or return immediately if the result has already arrived
    x = future1.get()
    y = future2.get()

    # call 'apply' and block until result arrives
    z = my_array[10,10].apply(5, ret=True).get()

def apply(self, x):
    self.data += x           # apply parameter
    return self.data.copy() # return result to caller
```

The `get` method of a future will block the thread on the caller side while it waits for the result, but it is important to note that it does not block the whole process. Other available work in the process (including of the same chare that

blocked) will continue to be executed.

*2. With remote method invocation:*

```
# --- in chare 0 ---
def work(self):
    group[1].apply(3) # tell chare 1 to apply 3 to its data, returns immediately

def storeResult(self, data):
    # got resulting data from remote object
    # do something with data

# --- in chare 1 ---
def apply(self, x):
  self.data += x  # apply parameter
  group[0].storeResult(self.data.copy())  # return result to caller
```

## 5.4 Reductions 101

Reductions can be performed by members of a collection with the result being sent to any chare or future of your choice.

```
# reduction.py
from charm4py import charm, Chare, Group, Reducer

class MyChare(Chare):

    def work(self, data):
        self.contribute(data, Reducer.sum, self.thisProxy[0].collectResult)

    def collectResult(self, result):
        print("Result is", result)
        exit()

def main(args):
    my_group = Group(MyChare)
    my_group.work(3)

charm.start(main)  # call main([]) in interactive mode
```

In the above code, every element in the group contributes the data received from main (int of value 3) and the result is added internally by Charm and sent to method `collectResult` of the first chare in the group (to the chare in processor 0 because Groups have one chare per PE). Chares that are members of a collection have an attribute called `thisProxy` that is a proxy to said collection.

For the above code, the result of the reduction will be 3 x number of cores.

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 finish.

---

**Hint:** Reductions are highly optimized operations that are performed by the runtime in parallel across hosts and processes, and are designed to be scalable up to the largest systems, including supercomputers.

---

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`, `gather`), as well as allowing users to easily define their own custom reducers for use in reductions. Please refer to the manual for more information.

Arrays (array.array) and NumPy arrays can be passed as contribution to many of Charm4py's built-in reducers. The reducer will be applied to elements having the same index in the array. The size of the result will thus be the same as that of each contribution.

For example:

```
def doWork(self):
    a = numpy.array([0,1,2])  # all elements contribute the same data
    self.contribute(a, Reducer.sum, target.collectResult)

def collectResult(self, a):
    print(a)  # output is array([0, 4, 8]) when 4 elements contribute
```

## 5.5 Hello World

Now we will show a full *Hello World* example, that prints a message from all processors:

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

class Hello(Chare):

    def SayHi(self):
        print("Hello World from element", self.thisIndex)

def main(args):
    # create Group of Hello objects (one object exists and runs on each core)
    hellos = Group(Hello)
    # call method 'SayHi' of all group members, wait for method to be invoked on all
    hellos.SayHi(ret=True).get()
    exit()

charm.start(main)  # call main([]) in interactive mode
```

The *main* function requests the creation of a `Group` of chares of type `Hello`. As explained above, group creation is asynchronous and as such the chares in the group have not been created yet when the call returns. Next, *main* tells all the members of the group to say hello, and blocks until the method is invoked on all members, because we don't want to exit the program until this happens. This is achieved by requesting a future (using `ret=True`), and waiting until the future resolves by calling `get`.

When the `SayHi` method is invoked on the remote chares, they print their message along with their index in the collection (which is stored in the attribute `thisIndex`). For groups, the index is an `int` and coincides with the PE number on which the chare is located. For arrays, the index is a `tuple`.

In this example, the runtime internally performs a reduction to know when all the group elements have concluded and sends the result to the *future*. The same effect can be achieved explicitly by the user like this:

```
# hello_world2.py
from charm4py import Chare, Group, charm

class Hello(Chare):

```

(continues on next page)

(continued from previous page)

```
    def SayHi(self, future):
        print("Hello World from element", self.thisIndex)
        self.contribute(None, None, future)

def main(args):
    # create Group of Hello objects (one object exists and runs on each core)
    hellos = Group(Hello)
    # call method 'SayHi' of all group members, wait for method to be invoked on all
    f = charm.createFuture()
    hellos.SayHi(f)
    f.get()
    exit()

charm.start(main)  # call main([]) in interactive mode
```

As we can see, here the user explicitly creates a future and sends it to the group, who then initiate a reduction using the future as reduction target.

Note that using a reduction to know when all the group members have finished is preferable to sending multiple point-to-point messages because, like explained earlier, reductions are optimized to be scalable on very large systems, and also simplify code.

This is an example of the output of Hello World running of 4 processors:

```
$ python -m charmrun.start +p4 hello_world.py ++quiet
Hello World from element 0
Hello World from element 2
Hello World from element 1
Hello World from element 3
```

The output brings us to an important fact:

---

**Note:** For performance reasons, by default Charm does not enforce or guarantee any particular order of delivery of messages (remote method invocations) or order in which chare instances are created on remote processes. There are multiple mechanisms to sequence messages. The `when` decorator is a simple and powerful mechanism to specify when methods should be invoked.

---

# CHAPTER 6

# Core API

## 6.1 charm

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

  Start the runtime system. This is required in *all* processes, and registers chare types 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 *one* PE (typically 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's arguments.

  If `interactive` is `True`, an entry point is not needed and instead Charm4py will transfer control to a Read-Eval-Print Loop (REPL) loop on PE0.

  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 explictly using the `classes` optional parameter. These must be references to the classes that are to be registered, and can reference classes in other modules. The `modules` parameter can be used to automatically register any chare types defined in the specified list of modules. These 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(exitCode=0)**:

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

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

  Makes the calling thread block until all of the chares in the collections referenced by the given proxies have been created on the system.

  **Note:** This can only be called from within the context of a *threaded method*, and the caller must have initiated the creation of the collections.

- **charm.createFuture(senders=1)**:

  Create and return a *Future*. The caller must be running within the context of a *threaded method*.

- **charm.getTopoTreeEdges(pe, root_pe, pes=None, bfactor=4)**:

  Returns a tuple containing the parent PE and the list of children PEs of `pe` in a tree spanning the given `pes`, or all PEs if `pes` is `None`. If `pes` is specified, `root_pe` must be in the first position of `pes`, and `pe` must be a member of `pes`. `bfactor` is the desired branching factor (number of children of each PE in the tree).

  In most systems, the resulting tree should be such that a physical node or host will have only one incoming edge (from parent).

- **charm.printStats()**:

  Print profiling metrics and statistics. `charm4py.Options.PROFILING` must have been set to `True`.

## 6.2 Chare

An application specifies new chare types by defining classes that inherit from `charm4py.Chare`. These classes can have custom attributes and methods. In addition, every chare instance has the following properties:

### 6.2.1 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.

### 6.2.2 Methods

- **contribute(self, data, reducer, target)**:

  Contribute data for a reduction operation across the elements of the collection to which the chare belongs to. The method must be called by all members to perform a successful reduction. `data` is the data to reduce; `reducer` is the reducer function to apply (see *Reducer*); `target` will receive the result of the reduction. The target can be the remote method of any chare(s) (indicated by `proxy.method`) or a *Future*.

- **wait(self, condition)**:

  Pauses the current thread until the specified condition is true. `condition` is a string containing a Python conditional statement. The conditional statement can reference attributes of the chare, constants and globals, but not local names in the caller's frame. To use this construct, the caller must be running within the context of a threaded method (see below).

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

  Requests migration of the chare to the specified PE. Note that this should be the last instruction executed by the chare's current call stack. The chare must be *migratable*.

### 6.2.3 Remote methods (aka entry methods)

Any user-defined methods of the chare type can be invoked remotely (via a *Proxy*). 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.

#### "threaded" method decorator

Methods tagged with this decorator will run in their own thread when called via a proxy. This allows pausing the execution of the method to wait for certain events (see `wait` construct above, *Futures* or `charm.awaitCreation()`).

The decorator is placed before the definition of the method, using the syntax: `@charm4py.threaded`

---

**Note:** While a thread is paused, the runtime continues scheduling other work in the same process, even for the same chare.

---

---

**Important:** The application entry point is always threaded.

---

#### "when" method decorator

The semantics of when a remote method can be invoked *at the receiver* can be controlled using the `when` decorator. The decorator is placed before the definition of the method, using the syntax:

`@charm4py.when('condition')`

where `condition` is a string containing a standard Python conditional statement. The statement can reference any of the chare's attributes (prefixed by `self`), as well as any of the method's arguments (referenced by their name).

---

**Important:** Callers are free to invoke the method whenever they are ready, without having to wait for the receiver (callee) to be ready. The message will be delivered and buffered at the receiving side until the receiver is ready and the condition is met. This is desirable for performance reasons.

---

## 6.3 Proxy

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 can be sent to any chare(s) in the system via remote methods.

---

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, ret=False)**:

> Calls the method of the chare(s) referenced by the proxy. This represents 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 the optional keyword argument `ret` is `True`, this 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 `None` and will be returned when the method has been invoked on every element.

Proxies that reference a Group/Array or its elements have additional properties (see *Group* and *Array*).

## 6.4 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:

`charm4py.Group(chare_type, args=[])` where `chare_type` is the type of chares that will constitute the group. `args` is the list of arguments to pass to the constructor of each element.

The call to create a group returns a proxy to the group.

Any number of groups (of the same or different chare types) can be created. Each group that is created has a unique integer identifier, called the "Group ID".

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

### 6.4.1 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 (see *Chare*). Like any proxy, group proxies can be sent to *any* chares in the system.

#### Attributes

- **gid**: The ID of the group that the proxy references.
- **elemIdx**: This is `-1` if the proxy references the whole group, otherwise it is the index of an individual element in the group.

#### Methods

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

## 6.5 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:

`charm4py.Array(chare_type, dims=None, ndims=-1, args=[], map=None)` 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.
2. Empty array of unspecified bounds, when `dims=None`. `ndims` indicates the number of dimensions to be used for indexes.

`args` is the list of arguments to pass to the constructor of each element. `map` can be optionally 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.

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. Each array that is created has a unique integer identifier, called the "Array ID".

**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.

### 6.5.1 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 (see *Chare*). Like any proxy, array proxies can be sent to *any* chares in the system.

#### Attributes

- **aid**: The ID of the array that the proxy references.
- **ndims**: Number of dimensions of the indexes used by the array.
- **elemIdx**: This is an empty tuple if the proxy references the whole array, otherwise it is the index of an individual element in the array.

#### 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`.

### 6.5.2 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.

## 6.6 Reducer

`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 (see *Chare*).

### 6.6.1 Reducers

`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.
- `nop`: This is used for empty reductions which do not contribute any data. Empty reductions are useful to know when all the chares in a collection have reached a certain point (e.g. synchronization purposes). Passing `None` as reducer in `contribute` calls has the same effect.
- `gather`: Adds contributions to a Python list, and sorts the list based on the index of the contributors in their collection.

### 6.6.2 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 `Chare.contribute()` calls. It must take two parameters `(data, contributor)`, where `data` is the data passed in a contribute 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 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`.

## 6.7 Futures

Futures are objects that act as placeholders for values which are unknown at the time they are created. A future is an instance of `charm4py.threads.Future`.

### 6.7.1 Creation

Futures can be returned when invoking remote methods (see *Proxy*). This allows the caller to continue doing work and wait for the value at the caller's convenience.

Futures can also be created explicitly by calling `charm.createFuture(senders=1)`, which returns a new future object accepting `senders` 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 the caller.

**Note:** Futures can only be created from threads other than the main thread (see threaded entry methods).

### 6.7.2 "Future" 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 used by the chare that created the future.

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

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

### 6.7.3 Future as reduction target

A future can be used as a reduction target (see *Chare*). In this case, the result of the reduction will be sent to the future.

## 6.8 readonlies

`charm4py.readonlies` is an object that serves as a container for data that the application wants to broadcast to every process after the "entry point" has executed. Attributes added to `readonlies` during execution of the entry point will become available in the `readonlies` instance of every process.

For example:

```
from charm4py import charm, Chare, Group
from charm4py import readonlies as ro

class Test(Chare):
    def __init__(self):
        # this will print 3 and 5 on every PE
        print(ro.x, ro.y)

def main(args):
    ro.x = 3
    ro.y = 5
    Group(Test)

charm.start(main)
```

**Warning:** Names of attributes added to `readonlies` must not start or end with '_'.

As the name implies, data in `readonlies` is intended to be read-only. This is because the broadcast is done only once, and after this there is no synchronization of the data.

## 6.9 Options

`charm4py.Options` is a global object with the following attributes:

- **PROFILING** (default=False): if `True`, charm4py will profile the program and collect timing and message statistics. To print these, the application must call `charm.printStats()`. Note that this will affect performance of the application.

- **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.
- **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 serialized). Best performance is obtained when this is enabled, but requires callers to relinquish ownership of any objects sent.
- **LOCAL_MSG_BUF_SIZE** (default=50): size of the pool used to store "local" messages (see above point).
- **AUTO_FLUSH_WAIT_QUEUES** (default=True): if `True`, messages or threads waiting on a condition (see "when" and "wait" constructs in *Chare* API) are checked and flushed automatically when the conditions are met. Otherwise, the application must explicitly call `self.__flush_wait_queues__()` of the chare.

CHAPTER 7

# Pool

The Charm 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. 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.

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 main function of `charm.pool` is currently parallel map. The syntax is:

- **charm.pool.map(function, iterable, ncores=-1, chunksize=1, allow_nested=False)**

  This is a parallel equivalent of the map function, which applies `function` to every item of `iterable`, returning the results. It divides the iterable into a number of chunks, based on the `chunksize` parameter, and submits them to the process pool, each as a separate task. This method blocks the calling thread 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 `allow_nested=True` if you want tasks to be able to spawn other parallel work (for example, tasks that themselves call `charm.pool`).

- **charm.pool.map_async(function, iterable, ncores=-1, chunksize=1, allow_nested=False)**

This is the same as the previous method but immediately returns a future, which can be queried asynchronously (see *Futures* for more information).

## 7.1 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

# Recursive Parallel Fibonacci

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

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

charm.start(main)
```

CHAPTER 8

# Performance

**Contents**



Python 3 with the Cython interface layer is currently the recommended way to run Charm4py programs to get the best performance (on CPython). This is the option that is installed when using pip.

## 8.1 Performance analysis

Set `charm4py.Options.PROFILING` to `True` to activate profiling. Do this this before the program starts.

`charm.printStats()`: prints timing results and message statistics *for the processor where it is called*. A good place to use this is typically right before exiting the program.

Here is some example output of `printStats()` for `examples/particle/particle.py` executed on 4 PEs:

```
Timings for PE 0:
|                    | em    | send  | recv  | total |
------ <class '__main__.Main'> as Mainchare ------
| collectMax         | 0.003 | 0.0   | 0.002 | 0.005 |
| done               | 0.0   | 0.0   | 0.0   | 0.0   |
------ <class '__main__.Cell'> as Array ------
| getNbIndexes       | 0.0   | 0.0   | 0.0   | 0.0   |
| resumeFromSync     | 0.518 | 0.009 | 0.001 | 0.528 |
| run                | 0.028 | 0.001 | 0.0   | 0.029 |
| updateNeighbor     | 4.942 | 0.113 | 0.069 | 5.124 |
```

(continues on next page)

(continued from previous page)

```
-----------------------------------------------------
|                   | 5.491 | 0.123 | 0.072 | 5.686 |
-----------------------------------------------------
| reductions        |       |       | 0.002 | 0.002 |
| custom reductions |       |       | 0.0   | 0.0   |
| migrating out     |       |       | 0.0   | 0.0   |
-----------------------------------------------------
|                   | 5.491 | 0.123 | 0.074 | 5.689 |

Messages sent: 4556
Message size in bytes (min / mean / max): ['0', '395.842405618964', '3121']
Total bytes = 1.72 MB

Messages received: 4110
Message size in bytes (min / mean / max): ['0', '461.45182481751823', '3713']
Total bytes = 1.809 MB
```

The output first shows timings (in seconds) for each chare that was active on the PE. Each row corresponds to a different entry method, where "entry" refers to the method being used as the target of a remote method invocation (via a proxy). Note that if the entry method itself calls other functions and methods locally (without a proxy), the time to execute those will be added to its own timings.

Timings are shown in four colums:

**em** Time in user code (outside of runtime) executing an entry method.

**send** Time in proxy and contribute calls (in runtime).

**recv** Time between Charm4py layer receiving a message for delivery and the target entry method being invoked (in runtime).

**total** Sum of the previous three columns.

The last rows show miscellanous overheads pertaining to reductions and migration.

---

**Note:** While all of the Charm4py code is instrumented, there are parts of the C/C++ runtime that are not currently reflected in the above timings.

---

The last part of the output shows message statistics for remote method invocations (number of messages sent and received and their sizes).

Charm++ has powerful tracing functionality and a performance analysis and visualization tool called *Projections*. This functionality has not yet been integrated into Charm4py.

## 8.2 Serialization

In many cases a remote method invocation results in serialization of the arguments into a message that is sent to a remote process. Serialization is also referred to as *pickling*. Pickling can account for much of the overhead of the Charm4py runtime. Fastest serialization is obtained with the C implementation of the `pickle` module (only available in CPython).

---

**Important:** Pickling can be bypassed for certain data and is encouraged for best performance (see next subsection).

---

A general guideline to achieve good 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.

### 8.2.1 Bypassing pickling

**This feature is currently only fully supported with Python 3 and Cython/CFFI**.

Best performance is achieved when passing arguments that support the buffer protocol (byte arrays, array.array and NumPy arrays). These bypass pickling altogether and are directly copied from their memory buffer in Python into a message in the Charm C++ library for sending. Note that these types of arguments can be freely intermixed with others not supporting the buffer protocol. For example:

```
particle1 = Particle(3, 5)     # Particle is a user-defined custom type
particle2 = Particle(7, 19)
A = numpy.arange(100)          # 100 element numpy array
proxy.work([1,2], particle1, A, particle2) # arguments 0, 1 and 3 will be pickled,
                                           # 2 will bypass pickling
```

CHAPTER 9

# Benchmarks

**Contents**



This section presents Charm4py benchmark results using: (a) real examples and miniapps; (b) synthetic test cases to evaluate specific features.

## 9.1 Mini-apps

### 9.1.1 LeanMD - Molecular Dynamics

We have ported the LeanMD Charm++ mini-app to Charm4py, with all the code written in Python. The physics functions are JIT compiled by Numba. The code is currently available in the `leanmd-experimental` branch of Charm4py.

Here we compare the performance between the C++ and Python versions. First we ran a *strong scaling* problem on Blue Waters with 8 million particles, obtaining the following results:

As we can see, the performance and scaling characteristics of Charm4py closely mimic the behavior of the C++ program. The y axis is logarithmic scale, and we can see that performance scales linearly with the number of cores. The average performance difference between Charm4py and Charm++ is 19%.

We also ran a different problem size (51 million particles) and configuration to evaluate performance with very high core counts (131k cores on Blue Waters), obtaining the following results:

| Version | Time per step (ms) |
|---|---|
| charm4py | 438 |
| Charm++ | 458 |

Here we can see that Charm4py performs better than C++. At this core count, parallel overhead becomes significant and Charm4py benefits from a feature that is not yet implemented in the C++ version of Charm. The feature allows aggregation of receives to local objects in the same *section*, thus reducing overhead.

## 9.2 Features

### 9.2.1 Bypass pickling for NumPy and other arrays

This feature is also known as "direct-copy". As explained in *Serialization*, it is used when NumPy arrays and other structures supporting the buffer protocol are passed as arguments of remote methods.

To test and evaluate the performance of this feature we wrote a small program (`tests/test_dcopy.py`) where a chare array is created, and each element sends three large data arrays to the rest of the elements, for a fixed number of iterations. The experiment was carried out using 4 cores on a standard Macbook Pro. The results are shown below (for 10 iterations):

| Metric | Without dcopy | With dcopy | speedup |
|---|---|---|---|
| Send time (s) | 2.406 | 1.046 | 2.3002 |
| Receive time (s) | 0.372 | 0.343 | 1.0845 |
| Total program time (s) | 12.72804 | 11.21846 | 1.1346 |
| Bytes sent (MB) | 1339.892 | 1339.323 | 1.0004 |

Note: this feature is enabled by default with Python 3 and Cython/CFFI layers.

CHAPTER 10

# Contact

You can contact us in the forum for discussion of any topics related to Charm4py or Charm++.

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

# CHAPTER 11

# Release Notes

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

## 11.1 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.

## 11.2 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.

## 11.3 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.

## 11.4 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.

## 11.5 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.

## 11.6 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

## 11.7 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