# CharmPy: A Python Parallel Programming Model

Juan J. Galvez, Karthik Senthil, Laxmikant V. Kale

Department of Computer Science
University of Illinois at Urbana-Champaign, IL, USA

E-mail: {jjgalvez, skk3, kale}@illinois.edu

***Abstract*—Parallel programming can be extremely challenging. Programming models have been proposed to simplify this task, but wide acceptance of these remains elusive for many reasons, including the demand for greater accessibility and productivity.**

**In this paper, we introduce a parallel programming model and framework called CharmPy, based on the Python language. CharmPy builds on Charm++, and runs on top of its C++ runtime. It presents several unique features in the form of a simplified model and API, increased flexibility, and the ability to write everything in Python. CharmPy is a high-level model based on the paradigm of distributed migratable objects. It retains the benefits of the Charm++ runtime, including dynamic load balancing, asynchronous execution model with automatic overlap of communication and computation, high performance, and scalability from laptops to supercomputers. By being Python-based, CharmPy also benefits from modern language features, access to popular scientific computing and data science software, and interoperability with existing technologies like C, Fortran and OpenMP.**

**To illustrate the simplicity of the model, we will show how to implement a distributed parallel map function based on the Master-Worker pattern using CharmPy, with support for asynchronous concurrent jobs. We also present performance results running stencil code and molecular dynamics mini-apps fully written in Python, on Blue Waters and Cori supercomputers. For stencil3d, we show performance similar to an equivalent MPI-based program, and significantly improved performance for imbalanced computations. Using Numba to JIT-compile the critical parts of the code, we show performance for both mini-apps similar to the equivalent C++ code.**

***Index Terms*—programming model, parallel programming, distributed computing, multiprocessing, Python, HPC**

## I. Introduction and motivation

Effective and productive programming of parallel machines can be extremely challenging. To this day, it remains hard to find programming models and frameworks that are considered accessible and productive by a wide range of users, support a variety of use cases, and achieve good performance and scalability on a wide range of systems. There is demand from programmers across various domains to write parallel applications, but they are neither computer scientists nor expert programmers. This often leads to their need to rely on experts to implement their ideas, settle for suboptimal (sometimes serial) performance, or to develop codes that are difficult to scale, maintain and extend.

A programming model must meet several demands to overcome these challenges, including: (a) accessibility (easy to approach, learn and use); (b) productivity; (c) provide high-level abstractions that can hide the details of the underlying hardware and network; (d) achieve good parallel performance; (e) make efficient use of resources in heterogeneous environments; (f) portability; (g) easy to integrate with existing software. The productivity of a language and programming model, in particular, can be a critical factor for the successful development of a software project, and for its continued long-term evolution.

In the realm of High-performance Computing (HPC), MPI combined with C/C++ or Fortran is widely used. Reasons for this include performance/scalability, the perceived sustainability of these technologies, and the existence of large legacy codebases. However, though important building-blocks, these technologies by themselves present important limitations towards achieving the above goals. C++ and Fortran are arguably not introductory-level programming languages. MPI provides message passing and synchronization primitives, but lacks high-level features like hardware abstractions, dynamic resource allocation, work scheduling; and is not particularly suited for execution of asynchronous events, or applications with load imbalance and irregular communication patterns.

Many parallel programming languages and runtimes have been developed in the last two decades [1], with modern ones providing high-level abstractions, task-based runtimes, global address spaces, adaptive load balancing, and message-driven execution. Examples of modern languages and runtimes include Chapel [2], X10 [3], UPC [4], Legion [5], HPX [6] and Charm++ [7]. In spite of this, MPI remains by all appearances the *de facto* standard for parallel programming in the HPC field. Analyzing the causes of this is outside the scope of this paper, but we believe that, although these models provide powerful abstractions, scalability and performance, obstacles for adoption include either a real or perceived lack of accessibility, productivity, generality, interoperability and sustainability. Charm++ has enjoyed success with several large applications running on supercomputers [8]–[10], but can be improved in terms of some of these aspects.

Parallel programming frameworks based on Python have emerged in recent years (e.g. Dask [11] and Ray [12]). Although aimed at productivity, they tend to have limited performance and scalability, and applicability only to specific use cases (e.g. task scheduling, MapReduce, data analytics).

In this paper, we introduce a general-purpose parallel programming model and distributed computing framework called CharmPy, which builds on Charm++, and is aimed at overcoming these challenges. One of its distinguishing features is that it uses the Python programming language, one of the most popular languages in use today [13] together with C, C++ and Java. Python has become very popular for scientific computing, data science and machine learning, as evidenced by software like NumPy, SciPy, pandas, TensorFlow and scikit-learn. It is also very effective for integrating existing technologies like C, Fortran and OpenMP code. Its popularity and ease of use [14] helps to avoid the barrier of adopting a new language, and enables straightforward compatibility with many established software packages. In addition, the development of technologies like NumPy [15], Numba [16], [17] and Cython [18] presents a compelling case for the use of Python as a high-level language driving native machine-optimized code. Using these technologies, it is possible to express a program in Python using high-level concepts, and have the critical parts (or even the bulk of it) be compiled and run natively.

CharmPy runs on top of Charm++ [7], [19], a C++ runtime, but it is not a simple Python binding for it. Indeed, CharmPy's programming model is simpler and provides unique features that simplify the task of writing parallel applications, while retaining the runtime capabilities of Charm++. For example, Charm++ developers have to write special interface files for each distributed object type; these files have to be processed by a special translator that generates C++ code. In addition, the Structured Dagger [20] language is often necessary for expression of control flow and message order. With CharmPy, all of the code can be written in Python, and no specialized language, preprocessing or compilation steps are necessary to run an application. CharmPy also benefits from high-level features of Python, like automatic memory management and object serialization.

With CharmPy, we want to meet the following goals:

- Simple, high-level programming model.
- Based on the widely used Python programming language, equipped with modern features and extensive libraries.
- General-purpose: supporting a wide range of applications, including those with embarrassingly parallel workloads, and complex scientific simulations running on supercomputers.
- High-performance: capable of achieving performance comparable to C++ parallel applications.
- Scalable from small devices to supercomputers.
- Programming and execution model with inherent communication and computation overlap.
- Adaptive runtime features, e.g. dynamic load balancing and automatic communication/computation overlap.

The achievement of some of these goals can be hard to quantify, relying in some cases on subjective assessment. We will present a use case to demonstrate the simplicity in terms of amount of code, readability and code complexity required to implement a non-trivial worker pool that can run multiple independent map functions on multiple nodes with dynamic load balancing, using the well-known master-worker pattern. We discuss the limitations of implementing the same use case with MPI. We will also show that parallel applications can be written with CharmPy that are comparable in terms of performance and scalability to applications using MPI or written in C++. This is possible even with applications fully written in Python, by using technologies like Numba. Python and Charm++ are both highly portable, and CharmPy runs on Unix, Windows, macOS and many supercomputer environments. The code is public and open-source [21].

The rest of the paper is organized as follows. In section II we explain the CharmPy programming model. Section III presents the parallel map use case. Section IV covers runtime implementation details. In section V we present performance results. Finally, in section VI we conclude the paper.

## II. The CharmPy programming model

In this section, we explain the main concepts of the CharmPy programming model, beginning with an overview of the programming paradigm and its execution model.

### A. Overview

CharmPy is based on the paradigm of distributed migratable objects with asynchronous remote method invocation. A program is expressed in terms of objects and the interactions between them. There can exist multiple distributed objects per processing element (PE); these objects can communicate with any other distributed object in the system via remote method invocation, which involves message passing. Objects are not bound to a specific PE and can migrate between PEs without affecting application code.

Parallel decomposition is therefore based on objects rather than system resources, which has the benefit of enabling more natural decomposition, abstraction from hardware, and gives the runtime flexibility to balance load, schedule work, and overlap computation and communication.

In the asynchronous execution model, a process does not block waiting for a remote method to complete, and the runtime can automatically continue scheduling other work during that time (which is facilitated by the presence of multiple objects per PE). Similarly, there are no blocking receives, and the runtime schedules delivery of messages as they become available. All of this serves to hide latency and enables the automatic overlap of communication and computation.

Another benefit of object-based decomposition, where an arbitrary number of objects per process can exist, is that it allows for tunable fine-grained decomposition without affecting the structure of the program. A fine-grained decomposition is particularly beneficial in irregular applications and those with load imbalance (the performance results in section V show an example of this).

The *execution model* on which CharmPy is based, including its benefits, has been described in detail and demonstrated in previous works [7]. In the rest of this section we will focus on the CharmPy programming model and API.

## B. Chare: the distributed object

In CharmPy, there is a class[1] named *Chare* that represents distributed objects. To define a new distributed object type, the user simply defines a new class that inherits from Chare, e.g. `class MyChare(Chare): ...`. Any methods of the new chare type will be callable remotely using regular Python method invocation syntax.

Chares can be created at any point after runtime initialization. The runtime is represented by an object called *charm* that exists on every process, and is initialized by calling `charm.start()`. After initialization, control is handed to the application via a user-defined function or chare, known as *entry point*, that runs on one processor (typically PE 0).

For example, the following is a complete program that creates a single chare, calls one of its methods, and exits after the method has been executed:

```
1   from charmpy import *
2
3   class MyChare(Chare):
4     def SayHi(self, msg):
5       print(msg)
6       charm.exit()
7
8   def main(args):
9     proxy = Chare(MyChare, onPE=-1)
10    proxy.SayHi('Hello')
11
12  charm.start(main)
```

This program can be run on multiple processes. Line 12 starts the CharmPy runtime on each process, and indicates that the function called `main` will be the entry point. In line 9, a single chare is created. The runtime can create the chare on any PE, because the application did not specify any. The call to create a chare returns a special object called *proxy*. Proxies are used to invoke methods remotely, and have the same methods as the chare that they reference. In this example, the method `SayHi` is called via the proxy. Remote method invocation is explained in detail in section II-D. Finally, the parallel program is finalized with a call to `charm.exit()`.

## C. Collections of chares

Chares can be organized into distributed collections. Collections are useful because they simplify the creation and management of sets of related chares, and enable efficient collective communication (broadcasts and reductions). There are two types of collections in CharmPy:

- *Arrays*: collections of chares indexed by keys, with members being able to exist anywhere on the system.
- *Groups*: where there is one member per PE.

The general syntax to create collections is:

```
proxy = Group(ChareClass, args=[x, y, ...])
proxy = Array(ChareClass, ..., args=[x, y, ...])
```

specifying the type of the collection (Group or Array), the chare class, and the arguments that will be passed to the constructor when members are instantiated. Array creation takes additional arguments (omitted above) to specify the type of index, initial size of the array, and optionally the initial mapping of chares to PEs (see section II-G for details on creating and managing chare arrays). An application can create multiple collections (of the same or different chare types).

It is important to note that in CharmPy, a given chare class can be used to create groups or any type of array. This differs substantially from Charm++, where a chare class is tied at declaration time to a specific type of collection. For example, in Charm++, a chare type declared to be part of a 3D-indexed array cannot be used to create single chares, groups, or arrays of index type other than 3D. No such restriction exists in CharmPy.

The following example shows how to create an array of $20 \times 20$ elements using 2D indexes:

```
proxy = Array(ChareType, (20,20))
```

In this example, the runtime decides how to distribute the chares among PEs, because a mapping has not been specified.

As we can see, creating a collection returns a *proxy*, which can be used to call methods of its members. This proxy references *all* of the members of the collection. As such, if a method is called on the proxy, it broadcasts the invocation to all members. Given a proxy to a collection and the index of an element, we can obtain a proxy to the individual element with: `element_proxy = proxy[index]`.

Chares that are part of a collection have two special attributes called *thisIndex* and *thisProxy*, the first is the index of the chare in the collection, and the second is a proxy to the collection.

## D. Remote method invocation

Proxies are used for remote method invocation. As we saw above, a proxy can reference a single object, or a collection of objects. Given a proxy, methods are invoked using standard Python function call syntax:

```
proxy.method(arg0, arg1, ...)
```

It is important to note that proxies can be passed to other chares as arguments of methods.

A method that is invoked on a chare as a result of a remote call (i.e. via a proxy), is also referred to as an *entry method*. Entry methods are invoked by message passing. If the caller and callee are not in the same process, the arguments are serialized[2] and sent in a message. If they are in the same process, however, the arguments are passed by reference directly to the callee, and a zero payload message will be sent instead. For this reason, the caller must give up ownership and not modify arguments after a remote method is invoked. This optimization between local chares is specific to CharmPy and applies to any type of entry method. In Charm++, a similar effect can be achieved by declaring *inline* entry methods, but the optimization is not applicable in general.

Calling remote methods returns immediately without waiting for the method to be executed at the remote object, and

[1] *Class* refers to the Object-oriented concept.

[2] For details on how arguments are serialized, refer to section IV-B.

consequently without waiting for a return value. Return values, if so desired, can be obtained in two ways: (i) via a separate method invocation if the receiver has a proxy to the caller; (ii) using futures. When invoking any remote method, a future [22] can be obtained by using the optional keyword argument *ret*:

```
future = proxy.method(args, ret=True)
```

The call returns immediately, and the caller can use the future to wait for the result at whatever time it is needed. Calling `future.get()` will return the value, blocking if it has not yet been received. For example:

```
result1 = remoteObj.getValue1(ret=True)
result2 = remoteObj.getValue2(ret=True)
# ... do additional work ...
# wait now for values from remoteObj
print('Result 1 is', result1.get())
print('Result 2 is', result2.get())
```

Futures can be used with broadcast calls also. Calling `get` will block until the method has been executed on all the chares of the collection. The return value will be `None`.

Is is important to note that blocking on a future does not block the entire process, and the runtime can continue scheduling other work (including for the same chare) while the caller is waiting. To use futures, the caller must be running in its own thread (see section II-H1). Futures are explained in more detail in section II-H3.

## E. Message order and dependencies

For performance reasons, the message-driven execution model of Charm++ does not guarantee by default that messages from chare $A$ to chare $B$ will be delivered in the same order in which they were sent. Also, unless otherwise specified, messages can be delivered at any time as soon as they become available at the receiving process.

There are situations, however, when messages have to be delivered in a certain order, or only when a receiver has reached a certain state, but relying on explicit synchronization between chares is undesirable for performance reasons. For these situations, CharmPy provides a simple and powerful construct that allows specifying when remote methods should be invoked at the receiver, in the form of the *when* decorator.

The decorator is placed before the method declaration, and its general syntax is:

```
@when('condition')
```

where condition is a string containing a standard Python conditional statement. The user can specify any general condition involving the chare's state and the arguments of the entry method. For example:

```
@when('self.x == x')
def myMethod(self, x, y, ...):
  # method is invoked when `self.x == x`
  ...
```

A common use case for this is to match messages for a method based on the current iteration in a simulation (in this case one argument would be the iteration to which the message belongs to, and the chare's attribute would be the chare's current iteration in the simulation).

With the "when" construct, CharmPy automatically buffers messages at the receiver and delivers them only when the user-specified condition is met. Because of this, a remote method can be called as soon as the caller is ready, without having to worry about messages arriving out of order, or the receiver not being ready, and avoids the need for explicit synchronization between chares. Additional examples are shown below:

```
@when('x + z == self.x')
def myMethod1(self, x, y, z):
  # method is invoked when the sum of the
  # first and third argument equal x attribute
  ...

@when('self.ready')
def myMethod2(self, arg0, arg1, ...):
  # method invoked when `self.ready` is True
```

For a future version of CharmPy, we are considering adding the capability of specifying when conditions on a per-message basis (at sender side). This would be optional and would not modify the existing API.

## F. Reductions

Reductions are one of the most common collective operations in CharmPy, and are used to apply a reduction function (also known as *reducer*) to a set of data that is distributed among the chares in a collection. CharmPy internally leverages the reduction framework in Charm++ to perform reductions in a distributed, asynchronous and scalable manner. The general syntax to perform a reduction is:

```
self.contribute(data, reducer, target)
```

To do a reduction, all of the chares in a collection must call this method. Here, *data* is the data contributed by the chare for reduction[3]. The *reducer* is the function that is applied to the set of contributed data. CharmPy provides several built-in reducers (including *sum*, *max*, *min*, *product*, *gather*), and allows users to easily define their own reducers. The *target* parameter determines who receives the result of the reduction. It can be a method of a chare or set of chares, specified using the syntax `proxy.method` where proxy can reference any individual chare or collection of chares (in the latter case the result is broadcast to all the members). The target can also be a future (see section II-H3).

It is worth noting that reductions are asynchronous, i.e. chares do not block waiting for reductions to complete, and there can be multiple reductions in flight (even for the same chare collection) at a given time.

An empty reduction can be performed by passing `data=None` and `reducer=None`. Empty reductions are useful for determining when a group of chares have reached a certain point in the application, and are typically used as a synchronization mechanism.

[3]In many cases *data* will be a NumPy array.

The following is a simple example of a sum reduction performed by members of a chare array, with the result being sent to element 0 of the array:

```python
class Worker(Chare):

  def work(self, data):
    data = numpy.arange(20)
    self.contribute(data, Reducer.sum,
                    self.thisProxy[0].getResult)

  def getResult(self, result):
    print("Reduction result is", result)
    charm.exit()

def main(args):
  # create 100 workers
  array = Array(Worker, 100)
  array.work()

charm.start(main)
```

*1) Custom reducers:* Users can also define their own reducer functions in CharmPy. The reducer function must take a single parameter which is a list of contributions (each from a different chare), and return the result of reducing the contributions. Registration of the reducer with CharmPy is done by calling `Reducer.addReducer(myReducerFunc)`.

### G. Chare arrays

The syntax to create N-dimensional chare arrays is:

`proxy = Array(ChareClass, dims, args)`

where *dims* is a tuple indicating the size of each dimension, and *args* the list of arguments passed to the constructor of every member. This will create $\prod_{i=0}^{|dims|-1} dims_i$ chares. N-dimensional arrays are indexed using Python n-tuples.

Arrays can also be sparse, i.e. a chare for every index in the index space need not exist. In this case, elements are inserted dynamically by the application. First, the array is created with the following syntax:

`proxy = Array(ChareClass, ndims=n, args)`

where $n$ is the number of dimensions of the index space. Elements can subsequently be inserted by calling:

`proxy.ckInsert(index, args)`

followed by a call to `proxy.ckDoneInserting()` when there are no more elements to insert. Custom indexes are also supported as long as they hash to a unique integer (by redefining Python's `__hash__` method).

*1) ArrayMaps:* When an array is created, the mapping of chares to PEs is by default decided by the runtime. This mapping can be customized using a built-in chare type called ArrayMap. The application can provide its own ArrayMap by defining a new chare class that inherits from ArrayMap and redefining the `def procNum(self, index)` method. This method takes an element index and returns the PE number where it should be created. A group of ArrayMap chares must be created prior to creating the array, and the proxy passed to the array creation function. For example:

```python
class MyMap(ArrayMap):
  def procNum(self, index):
    return index[0] % 20

def main(args):
  my_map = Group(MyMap)
  my_array = Array(MyChare, 10, map=my_map)
```

### H. Waiting for events

*1) Threaded entry methods:* In CharmPy, entry methods can run in their own thread when tagged with the `@threaded` decorator[4]. This allows pausing the execution of the entry method to wait for certain events. While the thread is paused, the runtime can continue scheduling other work in the same process. Threaded entry methods enable writing asynchronous applications in direct-style programming, simplifying expression of control flow. More specifically, it enables the two mechanisms described next.

*2) Wait for chare state:* The *wait* construct provides a convenient way to suspend execution inside a chare's entry method until the chare reaches a specific state. The syntax is:

`self.wait('condition')`

where condition is a string specifying a Python conditional statement, which is meant to evaluate the chare's state. Reaching this state will generally depend on the interaction of the chare with other chares. As an example, consider the following use case, where a chare performs an iterative computation. In each iteration, the chare sends data to a set of chares, waits to receive data from the same chares and subsequently performs a computation. The code is shown below:

```python
@threaded
def work(self):
  for iteration in range(NUM_ITERATIONS):
    for nb in self.neighbors:
      nb.recvData(...)
    self.wait(
        'self.msg_count == len(self.neighbors)')
    self.msg_count = 0
    self.do_computation()

def recvData(self, data):
  self.msg_count += 1
  # ... process data ...
```

Note that, to suspend control flow, the caller must be running within the context of a threaded entry method[5].

*3) Futures:* A future [22] is an object that acts as a proxy for a result that is initially unknown. Futures are present in many modern programming languages. In CharmPy, they are evaluated asynchronously, and will only block when the creator attempts to retrieve the value.

As explained in section II-D, futures can be used to query the result of remote calls. In addition, CharmPy allows futures to be created explicitly, sent to other chares, and be used to

[4]The main function or entry point is automatically threaded by default.
[5]CharmPy informs at run time if a method needs to be marked as *threaded*.

wait until an arbitrary operation (possibly involving multiple chares, and sequence of remote method invocations and reductions) has been completed. Chares in possession of the future can use it to send a value to the waiting chare. For example:

```python
@threaded
def start(self):
  f1 = charm.createFuture()
  f2 = charm.createFuture()
  remoteChare.doWork(f1, f2)
  # ... do other work ...
  print('Value of future 1 is', f1.get())
  print('Value of future 2 is', f2.get())

def doWork(self, f1, f2):
  # ...
  f1.send(x)   # send value to future 1
  # ...
  f2.send(y)   # send value to future 2
```

Futures can also be used as reduction targets:

```python
class Worker(Chare):
  def doWork(self, done_future):
    x = ...
    self.contribute(x, Reducer.sum, done_future)

def main(args):
  # create 100 workers
  workers = Array(Worker, 100)
  result = charm.createFuture()
  workers.doWork(result)
  # wait for work to complete and final reduction
  print('Reduction result is', result.get())
  charm.exit()

charm.start(main)
```

### I. *Chare migration*

Chares can migrate from one process to another. Reasons for migrating chares usually have to do with resource management, for example:

- Place communicating chares in the same process, host, or nearby host, to minimize latency and maximize bandwidth.
- Balance computational load, particularly if the workload of chares is not homogeneous.

The most common scenario is to let the runtime migrate chares where it considers appropiate, and the application only needs to signal the runtime at suitable points during execution at which to do this. However, chares can also be manually moved to a PE of the user's choice by calling: `self.migrate(toPe)`

The runtime takes care of serializing the object, migrating it to the new location and ensuring that messages directed to the object keep getting delivered [7]. CharmPy uses the pickle library to serialize chares, which by default attempts to serialize all of the attributes of the chare (which must be *pickable*). The application can customize what gets pickled (and in what form), by defining the `__getstate__` and `__setstate__` methods [23].

### J. *Automatic load balancing*

The Charm++ runtime has a load balancing framework that can automatically measure the load of chares, calculate a new assignment of chares to PEs (to balance load and/or optimize communication) and migrate the chares based on the obtained assignment. All of this occurs in a way that is transparent to the application. Possible ways to use this framework are for the application to tell the runtime when it is ready for load balancing, or by having the runtime perform load balancing periodically. CharmPy can leverage this framework, and we illustrate the benefits of load balancing in section V.

## III. USE CASE: DISTRIBUTED PARALLEL MAP WITH CONCURRENT JOBS

The *map* function is a feature in many programming languages that applies a given function to each element of a list, returning a list of results in the same order. Map can be implemented in parallel by applying the function to different inputs in parallel, collecting the results, and returning them to the caller. Python's multiprocessing library [24] implements a parallel version, however it is limited to a single node.

In this section, we will show how to implement a distributed parallel version that can run on multiple nodes using CharmPy. The implementation uses the well-known Master-Worker pattern. We will call each item of the input list a *task*. The master can coordinate multiple independent map jobs at the same time (which can be launched at different times), and is in charge of dynamically giving tasks to idle workers, thus ensuring that load balancing is achieved even in cases where tasks have disparate workloads.

First we show how the user-facing API looks:

```python
1  def f(x):
2    return x*x
3
4  def main(args):
5    pool = Chare(MapManager, onPE=0)
6    f1 = charm.createFuture()
7    f2 = charm.createFuture()
8    tasks1 = [1, 2, 3, 4, 5]
9    tasks2 = [1, 3, 5, 7, 9]
10   pool.map_async(f, 2, tasks1, f1)
11   pool.map_async(f, 2, tasks2, f2)
12   print("Final results are", f1.get(), f2.get())
13   charm.exit()
14
15 charm.start(main)
```

In this example, the main function creates a single master chare on PE 0, which will coordinate the pool of workers (line 5). It launches two separate jobs at the same time, to apply function $f$ to two different lists of items. A map job is initiated by calling the method `map_async` of the master, which receives the function to apply, the desired number of processors to use for the job, the list of tasks, and the future where the end result must be placed. It will block on line 12 waiting for the result of both jobs to arrive.

Now we will show how the master and workers are implemented[6]). A worker is a *chare* that has a `start` method used to signal the start of a new job. The method receives the job ID, the function to apply, the list of tasks in the job, and a proxy to the master. The code for workers is shown below:

```python
1  class Worker(Chare):
2
3    def start(self, job_id, f, tasks, master):
4      self.job_id = job_id
5      self.func   = f
6      self.tasks  = tasks
7      self.master = master
8      # request a new task
9      master.getTask(self.thisIndex, job_id)
10
11   def apply(self, task_id):
12     result = self.func(self.tasks[task_id])
13     self.master.getTask(self.thisIndex,
14                         self.job_id, task_id,
15                         result)
```

In addition, workers have a method called `apply` that receives a task ID and applies the function to the specified task (line 12). Subsequently, it calls the method `getTask` of the master (line 13) to request a new task. The arguments are the chare's index in the collection of workers (used to identify the worker), the job ID, and the ID and result of the last executed task (note how the previous result is sent at the same time as a new task is requested).

Next, we show the code for the master:

```python
1  class MapManager(Chare):
2
3    def __init__(self):
4      # create a Worker in every processor
5      self.workers = Group(Worker)
6      self.free_procs = set(range(1,
7                            charm.numPes()))
8      self.next_job_id = 0
9      self.jobs = {}
10
11   def map_async(self, func, numProcs, tasks,
12                 future):
13     """start a new map job"""
14     # select free processors
15     free = [self.free_procs.pop()
16             for i in range(numProcs)]
17     job = self.addJob(tasks, free, future)
18     # tell workers in free processors to start
19     for p in free:
20       self.workers[p].start(job.id, func, tasks,
21                             self.thisProxy)
22
23   def getTask(self, src, job_id, prev_task=None,
24               prev_result=None):
25     """called by worker to get a new task"""
26     job = self.jobs[job_id]
27     if prev_task is not None:
28       job.addResult(prev_task, prev_result)
29     if not job.isDone():
30       next_task = job.nextTask()
31       if next_task is not None:
32         self.workers[src].apply(next_task)
33     else:
34       for p in job.procs: self.free_procs.add(p)
35       self.removeJob(job)
36       job.future.send(job.results)
```

When the master is instantiated, it creates one worker on every PE by using a Group (line 5). It also initializes a set that is used to track which processors are not executing any job (line 6). When a new map job is requested via the `map_async` method, the master selects the requested number of free processors, stores the job information and signals the workers on the selected processors to start. Workers will begin requesting tasks by calling the method `getTask` of the master. This method simply stores the result provided by the worker, and sends a new task if there are more tasks to complete. If the job is done, it sends the result to the future that was provided by the user.

The full program is less than 100 lines of code and is available in the CharmPy source code repository.

Although a functionally equivalent program can technically be implemented using MPI, it requires a non-trivial amount of low-level code, more so if we want an API that is similar in terms of simplicity. Required low-level code would include: (a) asynchronous communication; (b) ability to receive messages of different types, from any source, at any time, and deliver to multiple destinations in a given process; (c) threads.

To illustrate this, note how PE 0 executes both the *main* function and the Master chare. At one point, the main function is waiting for two types of events (to receive the value of futures f1 and f2, being unknown which one will arrive first or from which process). At the same time, the master chare needs to process messages from any worker asking for new tasks, and could also receive messages asking to start new jobs (map_async).

This requires a non-blocking receive loop that can process any message type from any source, increasing the amount of low-level code needed. When a message is received, the type of the message and destination object needs to be determined, the arguments unpacked, and passed to the correct destination. Because communication must be non-blocking, and messages can be processed at different points in the code, threads are also required if we want application code to be able to wait until a certain event happens without exiting a function (see for example lines 10-12 in the main function).

CharmPy handles all of this complexity transparently, hiding it from the application.

[6]Trivial bookkeeping code is omitted for brevity.

## IV. Implementation details

CharmPy is implemented as a collection of Python modules that access the Charm++ runtime. In this section, we provide implementation details on selected parts of CharmPy.

### A. Parallelism

CharmPy programs are launched in the same way as many existing parallel applications (for example, those based on MPI and Charm++) that launch multiple processes on clusters and supercomputers. In this case, the executable that is launched is the Python implementation (e.g. CPython, Intel Python or PyPy), and the CharmPy program is passed as an argument.

The most common Python implementations do not support *concurrent* threads due to the restriction imposed by the Global Interpreter Lock (GIL) [25]. This does not represent a problem towards achieving parallelism, because we can run multiple CharmPy processes, which is also a typical use case with MPI. Chares running on different PEs will do so in different processes and execute concurrently.

It is important to note that the GIL restriction need not affect high-performance application code, and that efficient multithreading is possible for any code that runs outside of the interpreter. There are many common use cases for this in CharmPy, e.g.: NumPy can internally use multithreaded libraries; Intel Python uses MKL to accelerate NumPy, scikit-learn and other libraries; Numba can generate native multi-threaded code; and OpenMP code can be called from Python.

### B. Serialization

Remote methods are invoked like regular Python methods, with the arguments serialized and packed into a message. Arguments that are NumPy arrays, or other array structures, with contiguous memory layouts are copied directly from the object's memory buffer into the message. For these types of arguments, CharmPy includes metadata in the message header, to allow rebuilding them at the destination.

For all other types, CharmPy uses the *pickle* module [23] to serialize method arguments. Pickle can automatically serialize most Python types, including user-defined types. In addition, pickling of user-defined types can easily be customized, to decide what gets pickled and in what form [23]. In some situations, the overhead imposed by pickling can be too high (this is particularly true in fine-grained scenarios when pickling user-defined types). For the application critical path, we recommend using built-in data types and NumPy arrays (these bypass pickling altogether) when invoking remote methods.

Similarly, chares are also pickled when they migrate (explained in section II-I).

### C. Message passing

When a remote method is called, the arguments are serialized and packed into a message that is passed to the Charm++ library. Scheduling and inter-process communication is handled by Charm++. Messages are sent from the process where the source object is located to the process with the destination object. When a message reaches the destination process, it goes through the scheduler and delivered to the CharmPy runtime, which takes care of unpacking and deserializing the message and invoking the corresponding Python method.

As explained in section II-D, if the source and destination objects are on the same process, the data is not copied into a message and instead is passed by reference directly into a buffer of the destination object. Note that this differs from the general behavior of Charm++ and is made possible by the Python runtime.

Charm++ provides many communication layers, including one based on MPI, as well as layers for special interconnects like Intel OFI, Cray GNI and IBM PAMI.

### D. Reductions

Reductions involving common functions (like sum, max and min) with primitive data types (e.g. integer and floating point numbers) are performed entirely in C++ by the Charm++ runtime. In this case, the data is passed from CharmPy to the Charm++ library, which performs the reduction, asks CharmPy to convert the result to a CharmPy-compliant message, and sends the result to the target. Charm++ internally performs reductions in a distributed fashion using topology-aware spanning trees. For custom reducers, reductions are performed in a similar way, with the distinction that Charm++ transfers control to CharmPy when the user-defined reduction function needs to be applied.

### E. Cython implementation of CharmPy

Python modules can be implemented in C as so called C-extension modules. These interact with the Python interpreter at the C-level, are compiled, and can be used from code written in Python. Cython [18] generates C-extension modules from Python and Pyrex language, which is a superset of Python that allows for specific C constructs (particularly C types).

We have recently implemented critical parts of CharmPy in Cython, and will continue to extend this to other parts, as well as make further use of low-level optimizations. Note that all of this is transparent to application-level code and does not affect CharmPy's programming model or API.

## V. Performance evaluation with parallel scientific workloads

In this section we evaluate the performance of CharmPy on two different supercomputers: Blue Waters [26], a Cray XE system with AMD 6276 "Interlagos" processors and 3D torus topology; and Cori [27], a Cray XC40 system with Intel Xeon Phi "Knight's Landing" (KNL) nodes and Dragonfly topology.

We run two mini-apps: a 7-point stencil code on a 3D grid (referred to as *stencil3d*), and a molecular dynamics (MD) miniapp called *LeanMD* [28]. We have implemented versions of both mini-apps that are fully written in Python, with their computation-heavy functions compiled at runtime using Numba[7] [17]. As we will see, the performance achieved is similar to the equivalent C++ code. For stencil3d, we also compare performance with an MPI version of the code.

[7]All that is required is writing `@numba.jit` before the function declaration.

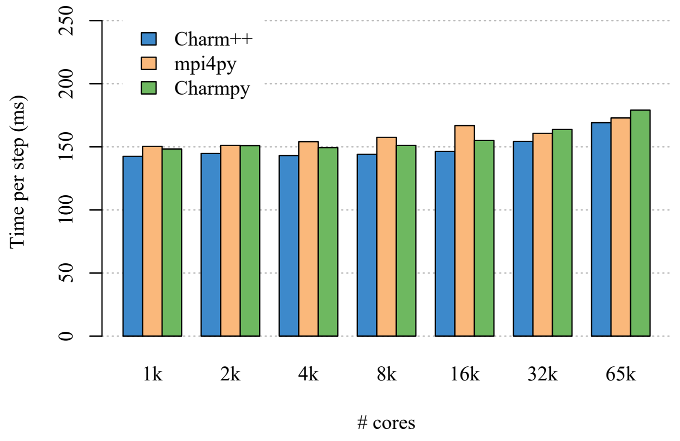

Fig. 1. Performance of stencil3d on Blue Waters (weak scaling) on up to 2048 nodes (using 32 cores per node), and 1000 iteration run.

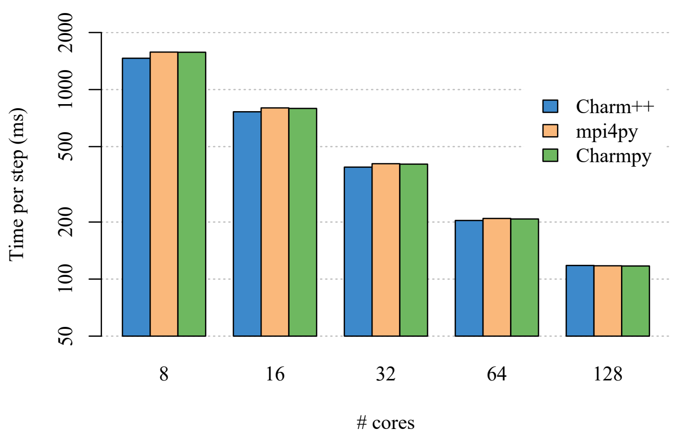

Fig. 2. Performance of stencil3d on 2 Cori KNL nodes (strong scaling) on 1000 iteration run. The y-axis scale is logarithmic.

## A. *Stencil3d*

The stencil3d benchmark implements a 7-point stencil code on a 3D grid decomposed into equal-sized blocks. The code is part of the CharmPy source code repository [21]. For this experiment, we also implemented an MPI version using mpi4py [29], which shares code with the CharmPy version. The compute kernel is the same in both versions, and is JIT-compiled at runtime using Numba/LLVM. The MPI code is thus similar to an MPI+C version, but note that there is an advantage over precompiled code in that the exact block size is known at JIT-compile time, and thus the quality of loop unrolling by LLVM can improve.

For MPI, we decompose the grid into $N$ blocks, where $N$ is the number of MPI ranks. CharmPy can use any arbitrarily fine-grained decomposition, i.e. the grid can be decomposed into any number of blocks (where each block is a chare), and there can be multiple blocks per process. The level of decomposition can be tuned without altering the structure of the program. For this experiment, however, we use the same decomposition as MPI (one block (chare) per process) because this is a highly regular application with no load imbalance.

Fig. 1 shows results of a weak scaling scenario on Blue Waters on up to 2048 nodes. As we can see, the performance of all three implementations is similar. In the worst case, the performance of CharmPy is only 6.2% lower than Charm++ (with 32k cores). The performance of the Charm++ version performs best in all cases, which is expected since it is fully implemented in C++, whereas the other two versions run part of their code in Python. It is worth noting that one of the main reasons why the performance of the Python versions is similar to C++ is because the compute kernel and other numerical sections (which are written in Python), are compiled to machine-optimized code using Numba. Based on the results, we can also see that CharmPy does not add a significant amount of runtime overhead compared to MPI or Charm++.

Fig. 2 shows a strong scaling scenario on 2 KNL nodes of Cori, scaling from 8 to 128 cores. Performance scales linearly from a time-per-step of roughly 1600 ms to 110 ms (y-axis scale is logarithmic). We observe very similar performance between the three implementations.

## B. *Stencil3d with load imbalance*

To show the dynamic load balancing capabilities of CharmPy, we modified the stencil3d benchmark to simulate load imbalance by having blocks perform different amount of work. This is achieved by extending the duration of the compute kernel by a factor $\alpha_i$ that varies depending on the block index $i$. That is, on each iteration the program measures the time to run the compute kernel (denoted as $t_k$), and waits for $t_k \times \alpha_i$ seconds. Let $N$ be the total number of blocks in the MPI version. Blocks with $i \leq 0.2N$ or $i \geq 0.8N$ always have a fixed load factor of $\alpha = 10$. For the rest, $\alpha_i = 100\frac{i}{N} + 5\lceil\frac{iter}{30}\rceil$, where $iter$ is the current iteration number. In practice, the nature of the imbalance is such that the ratio between the maximum load of a block and the average load of blocks is approximately 2.1 on average in the cases tested[8].

For CharmPy, we use a finer-grained decomposition of 4 blocks (chares) per process, required to be able to balance load by migrating chares. To ensure that the load of each portion of the grid is the same between MPI and CharmPy, the decomposition is made in a way that every chare is strictly within the confines of an MPI block, and chares that are part of the same MPI block $i$ have the same load factor $\alpha_i$.

Fig. 3 shows the results. For CharmPy and Charm++, we test with dynamic load balancing on and off ("lb" and "no lb", respectively). The application tells the runtime to balance load every 30 iterations. As we can see, the performance without load balancing is again very similar between the three implementations. With load balancing, performance improves substantially, ranging from 1.9x to 2.27x speedup.

[8]The average load is the lowest bound on the best possible maximum load after load balancing, and may or may not be achievable in practice depending on how load is quantized among blocks.

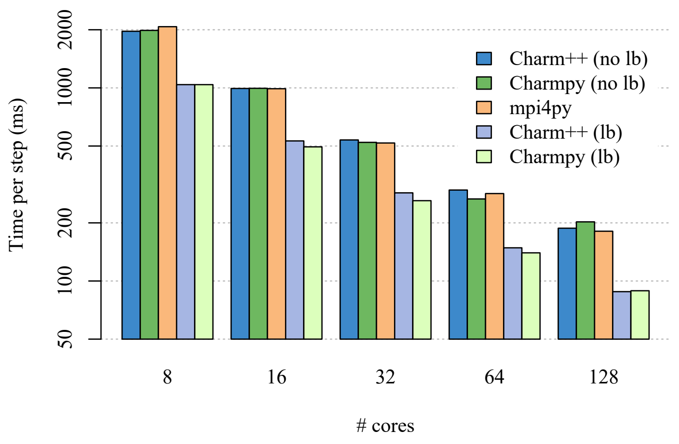

Fig. 3. Performance of stencil3d with synthetic imbalance, on 2 Cori KNL nodes (strong scaling) and 1000 iteration run. Performance with load balancing improves by a factor of up to 2.27x (y-axis logarithmic scale).

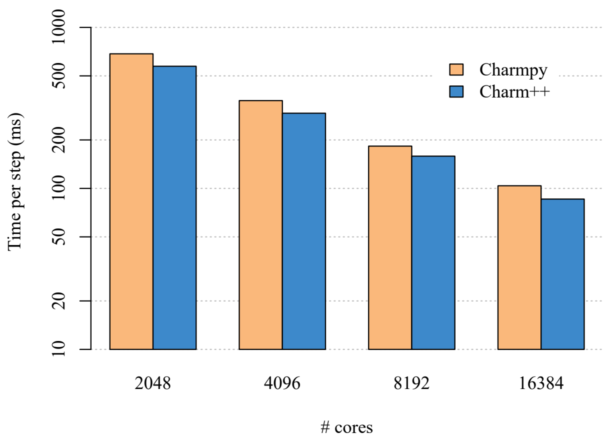

Fig. 4. Performance of LeanMD on Blue Waters with 8 million particles (strong scaling). The y-axis scale is logarithmic.

### C. LeanMD

LeanMD is a molecular dynamics simulation program for Charm++ that simulates the behavior of atoms based on the Lennard-Jones potential (which describes the interaction between two uncharged molecules or atoms). The computation performed in this code mimics the short-range non-bonded force calculation of NAMD [8], and resembles the LJ force computation of the miniMD benchmark in the Mantevo benchmark suite [30].

LeanMD uses a very fine-grained decomposition where a PE can have hundreds of chares at a given time, and there is simultaneous communication between many small groups of chares. We fully ported LeanMD to Python, with the physics code JIT-compiled by Numba.

Fig. 4 shows results running a strong-scaling benchmark on Blue Waters with 8 million particles. The performance of CharmPy is within 20% of the C++ version. This difference is higher than that observed in the previous benchmarks. The reason is the extra overhead of the CharmPy runtime, which is more pronounced in this case due to the large amount of chares per PE. We expect these results to improve in future versions of CharmPy, by making further use of Cython in performance-critical parts of the runtime, and heavier use of C-level optimizations (as explained in section IV-E).

## VI. Conclusion and Future Work

CharmPy is a parallel programming model based on the Python language and built on top of the Charm++ runtime. The model is based on the paradigm of distributed migratable objects with asynchronous message-driven execution. Adaptive runtime capabilities include dynamic load balancing and automatic overlap of computation and communication.

Our main design goals for CharmPy include simplicity, productivity, and the ability to take advantage of an efficient adaptive runtime system. In addition, we believe that providing a high-level parallel and distributed programming framework for Python, based on a proven model and runtime is important, as Python is quickly becoming very popular and its use widespread in areas like scientific computing, data science and machine learning. Distributed machine learning, for example, is an scenario that is being tackled by Ray (which shares some capabilities with CharmPy), and can benefit from the scalability and performance of CharmPy.

In this paper, we have explained the concepts and syntax of CharmPy, showing how to write parallel applications. One of the examples shown (implemented on less than 100 lines of code) is a general-purpose distributed map function that can run independent jobs simultaneously on multiple nodes, with load balancing support. We have also shown that it is possible to write parallel applications in Python using CharmPy that scale to very large core counts on supercomputers, and perform similarly to the equivalent MPI or C++ versions.

As future work, we plan to continue incorporating features from Charm++, including: fault-tolerance, shrink-expand, power and temperature optimizations, and upcoming support for heterogeneous computing. Shrink-expand (the capability to vary the amount of hardware resources during execution) is particularly useful in cloud environments. Heterogeneous computing will allow leveraging CPUs, GPUs and other accelerators on a node dynamically, balancing load between them. We are also planning on developing higher-level abstractions to distribute common Python workflows and data structures like NumPy arrays and pandas dataframes in a way that preserves their APIs.

## Acknowledgment

This research is part of the Blue Waters sustained-petascale computing project, which is supported by NSF award number OCI 07-25070 and the state of Illinois. Blue Waters is a joint effort of the University of Illinois and NCSA.

## References

[1] J. Diaz-Montes, C. Muñoz Caro, and A. Niño, “A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, pp. 1369–1386, 2012.

[2] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-oriented Approach to Non-uniform Cluster Computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 519–538.

[4] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and language specification," Tech. Rep., 1999, CCS-TR-99-157.

[5] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing Locality and Independence with Logical Regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012.

[6] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A Task Based Programming Model in a Global Address Space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. ACM, 2014.

[7] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice," ser. SC, 2014.

[8] P. J. C., B. Rosemary, W. Wei, G. James, T. Emad, V. Elizabeth, C. Christophe, S. R. D., K. Laxmikant, and S. Klaus, "Scalable molecular dynamics with NAMD," *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.

[9] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. Quinn, "Massively parallel cosmological simulations with ChaNGa," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008.

[10] N. Jain, E. Bohm, E. Mikida, S. Mandal, M. Kim, P. Jindal, Q. Li, S. Ismail-Beigi, G. J. Martyna, and L. V. Kale, "OpenAtom: Scalable Ab-Initio Molecular Dynamics with Diverse Capabilities," in *High Performance Computing*. Springer International Publishing, 2016, pp. 139–158.

[11] Dask Development Team, "Dask: Library for dynamic task scheduling," http://dask.pydata.org, 2016.

[12] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A Distributed Framework for Emerging AI Applications," *ArXiv e-prints*, Dec. 2017.

[13] "Interactive: The Top Programming Languages 2017," https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2017, IEEE Spectrum web article.

[14] P. F. Dubois, K. Hinsen, and J. Hugunin, "Numerical Python," *Computers in Physics*, vol. 10, no. 3, pp. 262–267, 1996. [Online]. Available: https://aip.scitation.org/doi/abs/10.1063/1.4822400

[15] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[16] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A LLVM-based Python JIT Compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. ACM, 2015, pp. 7:1–7:6.

[17] "Numba," https://numba.pydata.org/.

[18] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2011.

[19] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[20] L. V. Kale and M. Bhandarkar, "Structured Dagger: A Coordination Language for Message-Driven Programming," in *Proceedings of Second International Euro-Par Conference*, ser. Lecture Notes in Computer Science, vol. 1123-1124, September 1996, pp. 646–653.

[21] "CharmPy GitHub repository," https://github.com/UIUC-PPL/charmpy.

[22] B. Liskov and L. Shrira, "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, ser. PLDI '88. New York, NY, USA: ACM, 1988, pp. 260–267.

[23] "pickle - Python object serialization," https://docs.python.org/3/library/pickle.html.

[24] "multiprocessing - Process-based parallelism - Python 3.4.8 Documentation," https://docs.python.org/3.4/library/multiprocessing.html.

[25] D. Beazley, "Understanding the Python GIL," https://www.dabeaz.com/python/UnderstandingGIL.pdf, presented at PyCon 2010.

[26] National Center for Supercomputing Applications, "Blue Waters project," http://www.ncsa.illinois.edu/enabling/bluewaters.

[27] National Energy Research Scientific Computing Center, "Cori supercomputer," http://www.nersc.gov/users/computational-systems/cori/.

[28] "LeanMD," http://charmplusplus.org/miniApps/.

[29] L. Dalcin, R. Paz, M. Storti, and J. D'Elia, "MPI for Python: Performance improvements and MPI-2 extensions," *Journal of Parallel and Distributed Computing*, vol. 68, no. 5, pp. 655 – 662, 2008.

[30] "Mantevo Benchmark Suite," https://mantevo.org/.