Playdoh: a lightweight Python library for distributed computing and optimisation

Cyrille Rossant[a,b], Bertrand Fontaine[a,b], Dan F. M. Goodman[a,b]

[a] *Laboratoire Psychologie de la Perception, CNRS and Université Paris Descartes, Paris, France*
[b] *Département d'Etudes Cognitives, Ecole Normale Supérieure, Paris, France*

Abstract

Parallel computing is now an essential paradigm for high performance scientific computing. Most existing hardware and software solutions are expensive or difficult to use. We developed Playdoh, a Python library for distributing computations across the free computing units available in a small network of multicore computers. Playdoh supports independent and loosely coupled parallel problems such as global optimisations, Monte Carlo simulations and numerical integration of partial differential equations. It is designed to be lightweight and easy to use and should be of interest to scientists wanting to turn their lab computers into a small cluster at no cost.

Keywords: Python, parallel computing, distributed computing, optimisation, high performance computing

1. Introduction

In many areas of scientific research there is an increasing need for high-performance computing (HPC) [2, 15], witnessed by a large growth in publications relating to parallel or distributed algorithms for scientific computations (for example, [25, 5]). Until recently, HPC for scientific research has been restricted to a small number of areas and institutions with the required technological expertise and funding to exploit it. Recent trends in parallel and distributed computing are changing this picture, and now HPC is, in principle, accessible to almost all scientists thanks to several factors including: lower hardware costs; the ubiquity of relatively large numbers of powerful desktop computers in scientific labs; innovations such as general purpose graphics processing units (GPUs) [24] which provide parallel processing performance at a fraction of the cost of traditional clusters, and which are increasingly being used for scientific computations [16, 21]; and on-demand cloud computing services [30] such as Amazon EC2[1], Google App Engine[2], and PiCloud[3]. Although the resources are now widely available, the growth of scientific HPC is limited by the technical difficulties of using them. Traditional solutions adapted to the needs of the early users of scientific HPC (well-funded and with substantial technical expertise) are highly efficient, but are very difficult to use and can be expensive. In order to lower the technical barrier to entry of scientific HPC, and hopefully thereby spur the advances in science that should accompany universal access to HPC, there needs to be substantial improvements in the accessibility of software solutions so that scientists without expertise in parallel computing can make use of the resources at their disposal.

In this paper, we present Playdoh[4], a Python package offering a simple and small-scale grid computing solution to scientists. It allows for easy access to an abundant supply of existing parallel computing resources: the desktop computers already sitting on every desk in a scientific lab. Modern computers typically come with multiple processor cores (two, four or even six cores are already common), and GPU cards. A small lab of, for example, five researchers, each with a desktop computer equipped with a four core processor and a GPU card, already has access to formidable HPC resources which in most cases are left unused. Typically, computations in such a lab would make use of only one CPU core on only one of these machines, 1/20th of the resources available to them with CPUs alone and possibly 1/250th of the resources once GPUs are taken into account[5].

[1] http://aws.amazon.com/ec2/
[2] http://code.google.com/appengine/
[3] http://www.picloud.com/
[4] http://code.google.com/p/playdoh/
[5] Comparison based on 43 GFLOPS reported for the Intel i7 920 at 2.66GHz, and 1800 GFLOPS reported for the NVIDIA GeForce GTX 295

Playdoh is a lightweight package written in Python, an increasingly popular platform for scientific computing [23]. It has no dependencies except the widespread scientific library NumPy[6] making it easy to install and portable across most platforms. Playdoh presents a very simple and intuitive interface to the user, making it trivial to implement independent parallel (or "embarrassingly parallel") problems with a couple of lines of code, and straightforward to implement more sophisticated parallel computing tasks (loosely coupled problems, defined below). In addition, it features a global optimisation package focused on maximising fitness functions which are computationally expensive and do not have gradients. This sort of problem arises very frequently in scientific computation and can greatly benefit from parallel computing. Playdoh also supports distributing tasks written in CUDA[7] across GPUs on one or several computers.

There are many existing Python packages for parallel computing. In addition to including a package for distributed optimisation, Playdoh differs from them by offering a solution tailored to scientific computation in a lab with a small number of networked computers. It was originally designed for use in our lab, in particular for global optimisation problems applied to neural modelling [27, 28]. The limitation of the majority of existing packages to embarrassingly parallel problems was too restrictive for our needs, but on the other hand packages which went beyond this, including IPython, were based on the MPI framework, which seemed excessively complicated. In fact, IPython only uses MPI for coupled problems, for independent parallel problems it can be used without MPI. This related work is discussed in more detail in section 5. In addition, we wanted a package that allowed for dynamic allocation of resources so that each member of our group could always be guaranteed at least as many computational resources as were available on their machine, and that most of the time there would additionally be a pool of resources available on other machines.

In the next sections, we present how Playdoh can be used to execute independent parallel tasks and optimisations. We also show how more sophisticated parallel problems can be implemented by giving simple code examples of a Monte Carlo simulation and a partial differential equation numerical solver. We then briefly present the implementation of the core features of Playdoh. Finally, we give some performance results, and discuss the advantages and disadvantages of other existing parallel computation libraries.

2. Features

Playdoh allows for the distribution of computations across a pool of multicore computers connected inside a standard, for example, Ethernet based network (Figure 1). This kind of architecture is very common in scientific laboratories and is well adapted to independent and loosely-coupled parallel problems. Many computational tasks encountered in scientific research can be distributed in this way. The easiest and perhaps most common situation where a scientist could benefit from parallelism is when executing a set of independent jobs, for example when exploring a model in the parameter space. Playdoh, like many other libraries (see Section 5), provides a `playdoh.map` function which is syntactically similar to the built-in Python `map` function and allows for the distribution of independent jobs across CPUs on one or more computers. In addition, Playdoh handles more sophisticated cases of parallel computations where the subtasks are not absolutely independent but require infrequent low-bandwith communication [6]. Such problems are said to be *loosely coupled*. This category includes all problems that are spatially subdivided with only local interactions. Those problems can be distributed using domain decomposition techniques, where only the subdomains boundaries need to be exchanged. Frequent high-bandwidth communication between subtasks would not be efficient in the type of network architecture Playdoh is designed for, because data transfer becomes a major performance bottleneck. Many widespread types of computations can be distributed as such loosely-coupled parallel problems: Monte-Carlo simulations, optimisations, cellular automata, numerical solving of partial differential equations (PDE), etc.

Playdoh currently comes with a built-in global optimisation toolbox which contains several algorithms including the Particle Swarm Optimisation algorithm (PSO) [13, 29], a Genetic Algorithm (GA) [8, 17] and the Covariance Matrix Adaptive Evolution Strategy algorithm (CMA-ES) [12, 20]. The toolbox provides `minimize` and `maximize` functions to rapidly optimise a Python function in parallel. These global optimisation algorithms are best adapted to those with computationally intensive fitness functions for which the gradient is not available or does not exist.

[6] http://numpy.scipy.org/

[7] http://www.nvidia.com/object/cuda_home_new.html

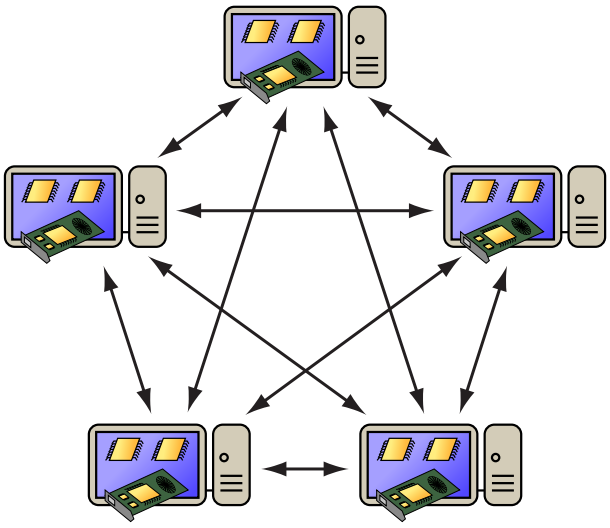

Figure 1: **Small-scale grid computing with Playdoh.** Playdoh allows the user to turn a set of interconnected computers into a small cluster. The CPUs (and GPUs if available) of the computers are shared with the other users, who can use these idle resources for their own computations. Each user can specify at any time the number of resources on their own computer to be allocated to the cloud, with the remainder being kept for their own usage.

For other sorts of problems, Playdoh offers a simple programming interface for implementing computations that can be distributed as loosely coupled parallel problems. In this section, we show with simple examples how all those problems can be implemented with Playdoh.

2.1. Independent parallel problems

The typical use case of the `playdoh.map` function is the parallel execution of a computational model with different parameters. In the following example, we show how to execute the two trivial operations 1 + 2 and 3 + 4 in parallel on two CPUs.

```
def sum(x, y):
    return x + y
import playdoh
print playdoh.map(sum, x=[1, 3], y=[2, 4], cpu=2)
```

2.2. Parallel optimisation

Global optimisation tasks are very common scientific computational problems. Here, we focus on situations where the major computational part of the optimisation is the evaluation of the (real-valued) fitness function rather than the optimisation algorithm itself. This scenario typically occurs when optimising a complex model with respect to some of its parameters, where the fitness depends upon the parameters in a potentially non-differentiable or even non-continuous way.

The following example shows how to minimize the square function $f(x) = x^2$ in parallel on two CPUs.

```
def square(x):
    return x ** 2
import playdoh
result = playdoh.minimize(square, x=[-10, 10], cpu=2)
print result
```

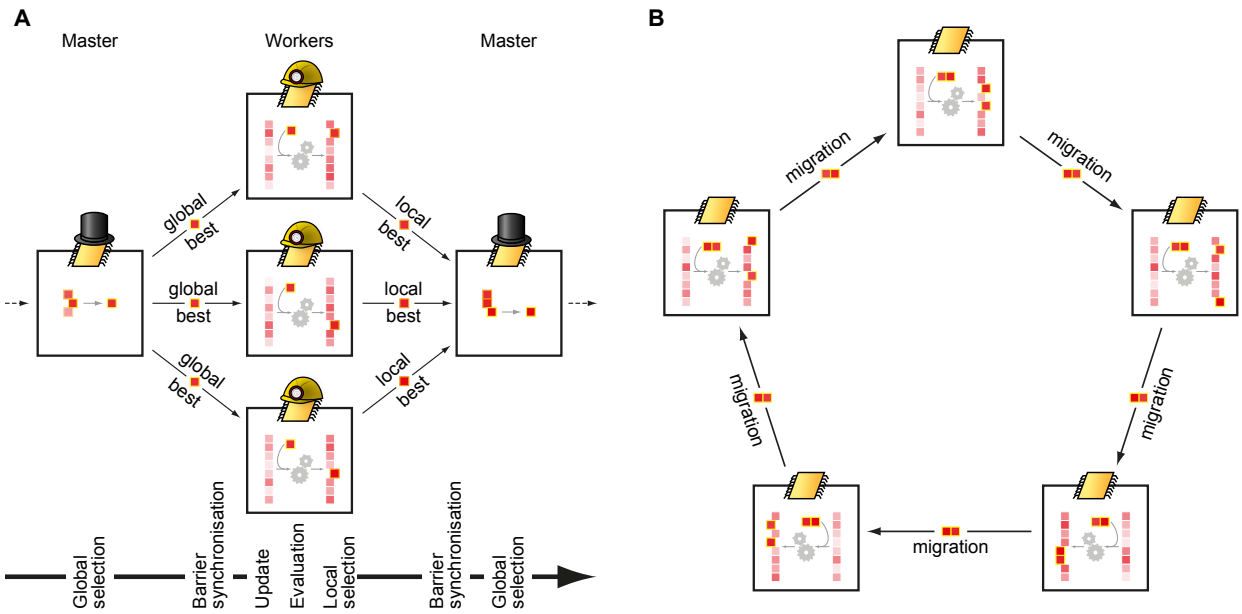

Figure 2: **Two parallel models for distributed optimisation. A.** Schematic illustration of the master-worker model used for the PSO algorithm. The workers keep their own set of particles which evolve according to the PSO rules. At every iteration, the workers send their local best particles to the master. Then, the master selects the best particle among those local best particles: this is the global best particle. The master finally sends this global best particle back to the workers. This model allows all nodes to know at each iteration the global best particle across all nodes. **B.** Schematic illustration of the island model used for the Genetic Algorithm. Every node (or island) keeps its own population, which evolves according to the GA rules. Every few iterations, the best individuals on each node migrate to the next node in a ring topology, and reproduce with the local population.

The `x` keyword argument here specifies the initial interval over which the particles should be uniformly sampled. Boundary conditions on the parameters can also be set here.

All the optimisation algorithms included in Playdoh consist of successive iterations of two steps. In the evaluation step, the fitness function is evaluated against a large number of parameters (points, or particles, in the parameter space). In the update step, the particles' positions evolve according to the optimisation algorithm rules. After a sufficient number of iterations, the particles are expected to converge towards the minimising or maximising position.

The evaluation step is independently parallel and can be easily distributed across different computing units (or nodes): each node evaluates the fitness function against a subset of the particles. Parallelising the update step involves the nodes exchanging the minimum information needed so that they all converge towards the same position.

Master-worker model in the PSO algorithm. In the PSO algorithm, the particles evolve towards a mixture of the global best position found so far and the local best position found so far by each particle. Therefore, distributing the algorithm implies the communication of the global best position to every node at each iteration. This can be done using the master-worker model: one chosen unit (the master) is responsible for finding and communicating the best position to every other unit (the workers) at each iteration (Figure 2A). This implies minimal data transfer since only a single particle position needs to be communicated.

Island model in the GA. In the GA, at every iteration, some particles (or individuals) are chosen based on their fitness and their parameters are crossed to generate a given percentage of the total population. Some other particles will mutate, i.e. some of their variables will be randomly resampled, and a small proportion of the best particle, the elite, will remain unchanged. The population is expected to converge towards the best position. This algorithm is distributed using the island model [31]: every unit (island) has its own population of particles which evolves independently from the others. Every few tens of iterations, the best particles of any island migrate to the next in a ring topology (Figure 2B). This has been shown to improve the performance of the optimisation. Only a small fraction of individuals migrate, and so data transfer is limited.

The optimisation toolbox is extensible in that new optimisation algorithms can be implemented using the parallel programming interface proposed by Playdoh. This interface can also be used to implement any other parallel computation, as shown in the next paragraph.

2.3. Loosely coupled parallel problems

Computational tasks that cannot be distributed using the independent parallel interface of Playdoh typically require some communication between subtasks and the introduction of synchronisation points. Playdoh offers a simple programming interface to let the user implement their computation by focusing on the parallelism logic for their specific task, without having to deal with the details of low-level inter-process communication. This programming interface is presented in the next two examples.

Parallel Monte Carlo simulation

In the first example we evaluate π using a classical Monte Carlo simulation (a very commonly used type of computation). We proceed by sampling random points in a square and counting the number of points inside the unit circle. If E_n is a set of n points uniformly sampled in $[0, 1]^2$ and $D = \{(x, y) \in E | x^2+y^2 \leq 1\}$, then $\pi_{MC} = 4\times|\{E_n \cap D\}|/n$ is an estimation of π.

For a Monte Carlo simulation such as this, it is straightforward to independently parallelise it: every node samples its own set of points and computes its own estimate. At the end, the estimations are combined to yield a more precise estimate. Data transfer happens only at the end of the task and is very limited in this example since only the count of points inside the unit circle is communicated. In fact, in this case the task would be best approached using `playdoh.map`, but we use it here to demonstrate the Playdoh programming interface. To implement this task, two stages are required. First, the task itself must be written: it is a Python class which actually performs the computation. Every computing unit stores and executes its own instance. Then, the task launcher executes on the client and launches the task on the CPUs on the local machine or on several machines across the network.

```
import numpy, playdoh
class PiMonteCarlo(playdoh.ParallelTask):
    def initialize(self, n):
        self.n = n  # number of points on this node
    def start(self):
        # Draws n points uniformly in [0,1]^2
        samples = numpy.random.rand(2, self.n)
        # Number of points inside the quarter unit circle
        self.count = numpy.sum(samples[0, :] ** 2 + samples[1, :] ** 2 <= 1)
    def get_result(self):
        return self.count
```

Launching the task is done by calling the Playdoh function `start_task`. Here, we launch the task on two CPUs on the local machine. The `start_task` function triggers the instantiation of the class on every node, the call to the `initialize` method with the arguments given in the `args` keyword argument, and finally the call to the `start` method.

```
points = [50000, 50000]  # number of points per unit
# <args> contains the arguments of <initialize> for every node
task = playdoh.start_task(PiMonteCarlo, cpu=2, args=(points,))
# Calls get_result on every node
result = task.get_result()
# Prints the estimation of Pi
print sum(result) * 4.0 / 100000
```

Parallel numerical solution of partial differential equations

Numerical solvers for partial differential equations (PDEs) are spatially subdivided and only have local interactions, which makes them good candidates for parallel computing in Playdoh. In this example, we show how to parallelize a numerical solver of the heat equation on a square with Dirichlet conditions using the finite difference method. If $S = [-1, 1]^2$, we write the equations as:

$$\forall t \in \mathbb{R}_+, \forall (x, y) \in S, \frac{\partial u}{\partial t}(t; x, y) = \Delta u(t; x, y)$$
$$\forall t \in \mathbb{R}_+, \forall (x, y) \in \partial S, u(t; x, y) = 0$$
$$\forall (x, y) \in S, u(0; x, y) = f(x, y)$$

This equation is numerically solved by discretising time and space using an Euler scheme (finite difference method) [19]. To parallelize this problem, we use a classic spatial domain decomposition technique [7, 18]. We divide the square grid into overlapping vertical bands: each node solves the equation on a band. At every iteration, the nodes need to communicate information about the boundaries of the bands to their neighbors (see Figure 3). The overlap is required to keep consistency between the subtasks. The size of the data transfer is only $O(\sqrt{n})$ per iteration, where n is the total number of points in the grid.

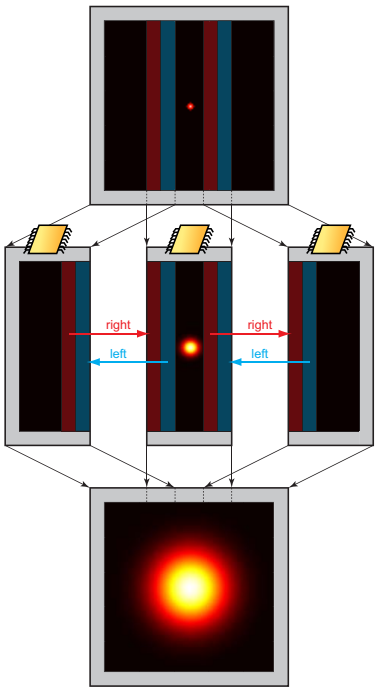

Figure 3: **Parallel numerical solving of a PDE.** This figure illustrates the parallel implementation of the PDE numerical solver example. The goal is to solve the heat equation inside a square with Dirichlet boundary conditions and with a centered Dirac initial function (top panel). The solution is expected to be a centered Gaussian function with increasing variance. The client decomposes the domain into vertical overlapping bands (top panel) and sends the decomposed initial function on these bands to the nodes. Each node solves the equation by computing a numerical approximation of the Laplacian on that band before using the forward Euler scheme. Computing the Laplacian requires the knowledge of the function on the boundaries, which are processed by the neighboring nodes. This is why the nodes send their boundaries to their neighbors (middle panel) at every iteration. At the end, the client retrieves the function values on every band and recombines them to obtain the final function values on the whole domain (bottom panel).

Communication between nodes happens through *tubes*, which are one-way named FIFO queues between two nodes. The source puts any Python object in the tube with a `push`, and the target gets objects in the tube with a (blocking) `pop`. This allows a simple implementation of synchronisation barriers. The following code example shows

the most relevant part of the task implementation: the computation involves iterative calls to `send_boundaries`, `recv_boundaries`, and `update_matrix`.

```
def send_boundaries(self):
    if 'left' in self.tubes_out:
        self.push('left', self.X[:, 1])
    if 'right' in self.tubes_out:
        self.push('right', self.X[:, -2])
def recv_boundaries(self):
    if 'right' in self.tubes_in:
        self.X[:, 0] = self.pop('right')
    if 'left' in self.tubes_in:
        self.X[:, -1] = self.pop('left')
def update_matrix(self):
    Xleft, Xright = self.X[1:-1, :-2], self.X[1:-1, 2:]
    Xtop, Xbottom = self.X[:-2, 1:-1], self.X[2:, 1:-1]
    self.X[1:-1, 1:-1] += self.dt * (Xleft + Xright + Xtop + Xbottom \
                          - 4 * self.X[1:-1, 1:-1]) / self.dx ** 2
```

The most relevant part in the task launcher is the definition of the topology (here, a double linear topology) as a list of tubes.

```
topology = [('right', 0, 1), ('left', 1, 0),...]}
```

The k nodes are uniquely identified with indices between 0 and $k-1$ and are mapped transparently at runtime to actual CPUs according to the available resources in the network.

2.4. Resource allocation

Resources (CPUs and GPUs) are shared with other computers on the network by running the Playdoh server with the `open_server` function. The server runs in the background and the computer can continue to be used by its operator, who can choose how many resources to reserve and how many to share. Resource allocation is done with a cross-platform graphical user interface included in Playdoh. When running a task on remote machines, the number of resources to use on every machine is calculated automatically at the beginning of the task (static allocation).

Resources include CPUs and GPUs: the latter can only be used if the CUDA code is provided along with the PyCUDA [14] code to launch it. In this case, several GPUs on a single machine or on different machines can be used in parallel.

3. Methods

Playdoh is coded in pure Python and uses only native Python modules except for NumPy which is a very standard library in the scientific community. The latest stable version (Playdoh 0.3.1) contains about 10,000 lines of code: about 20% are dedicated to inter-process communication, 15% to code transport, 15% to distributed computing, 30% to optimization, and the remaining 20% are related to the user-exposed functions, scripts and command-line tools.

3.1. Inter-process communication

The core component of Playdoh is a standalone and lightweight inter-process communication (IPC) module which lets processes on a single computer (possibly running on different processors) or on different interconnected machines to exchange information. This module, on which the other modules in Playdoh are based, implements a lightweight Remote Procedure Call (RPC) protocol: Python functions can be transparently called on a machine and executed on another one. A simple and transparent authentication procedure is implemented in this layer to ensure security of the communications. Each message is authenticated with a shared key using a hash-based message authentication code (HMAC). This module uses the native `multiprocessing`, `threading` and `cPickle` modules: computers communicate by serialising Python objects and sending them over TCP. Different CPUs are used through different processes rather than different threads: this is a known limitation due to the global interpreter lock (GIL) of CPython [3].

3.2. Shared memory

Nodes running on different computers need to have independent copies of data in memory, but nodes running on different CPUs on a same computer may have access to shared memory. With Playdoh, it is possible to store some NumPy arrays in shared memory. This can be more efficient than having in memory as many copies of one array as processes, especially with very large NumPy arrays. However, such shared arrays need to be read-only in order to avoid contention issues when several processes try to make changes to the same data at the same time.

3.3. Code transport

Running a task on different computers requires the task's code to be sent to every computer. This is done in Playdoh with a code serialization technique: the code of the function or the class implementing the task is automatically retrieved along with any Python modules which it depends on. The code is sent to the remote computers which can then execute the task.

4. Results

4.1. Independent parallel tasks

Performance tests were conducted to compare the speed improvement of distributing independent tasks using Playdoh with the maximal improvement expected from ideal parallelization. The tasks consist of a pause or a neural network simulation performed with the Brian simulator [9, 10], each one consisting of a typical network of 4000 neurons with 2% connectivity (the standard current based, or CUBA, network from [4]) for a simulation duration of 2 seconds. The pauses show the scaling for a perfectly homogeneous network where the computation takes the same time on each CPU. In Figure 4A, it can be seen that scaling is very good in this case, improving substantially as the length of the pause increases (corresponding to a more complex computation) relative to the inter-machine communication time. The scaling with the neural network simulation shows that in a more realistic situation where the computation time is different for different CPUs (because of heterogeneous computers), the scaling is not quite so close to ideal because the total time is controlled by the time for the slowest CPU. However, even in this case the scaling is still good, and the addition of load balancing to Playdoh (planned for a future release, see section 6) would bring it closer to parity with the performance for the pauses.

4.2. Optimisation performance

To illustrate the use of the optimisation algorithms, two widely used functions are implemented. The first one, the Rosenbrock function, has a global minimum inside a long, narrow, parabolic shaped flat valley and no local minima. Whereas it is trivial to find the valley, converging to the global minimum is difficult. The function is defined as follows:

$$f(x) = \sum_{i=1}^{n-1} \left[(1 - x_i)^2 + 100(x_{i+1} - x_i^2)^2 \right]$$

where n is the number of dimensions (we choose $n = 5$ here). The search domain is set to $-2.2 \leq x_i \leq 2.2, i = 1, 2, ..., n$ and its global minimum $f(x^*) = 0$ is at $x_i^* = 1, i = 1, 2, ..., n$. The second function used is the Schwefel function which has several local minima close to the global minimum. It is defined as

$$f(x) = 418.9829n + \sum_{i=1}^{n} (-x_i \sin(\sqrt{|x_i|}))$$

The search domain is set to $-500 \leq x_i \leq 500, i = 1, 2, ..., n$ and its global minima $f(x^*) = 0$, which is close to the boundaries, is located at $x_i^* = -420.9687, i = 1, 2, ..., n$.

For the Rosenbrock function the performance of the PSO algorithm is compared to that of the CMA-ES. In this example the number of particles is 80, the number of iterations 150 and the number of workers is 2. As can be seen in Figure 4D, the CMA-ES algorithm converges much more rapidly to the solution than the PSO. For the Schwefel, a GA on a single worker, i.e. without any islands, and a GA with 4 workers, i.e. with 4 islands, are implemented. In this example the number of particles is 1000 per worker and the number of iterations is 2000. The results, which

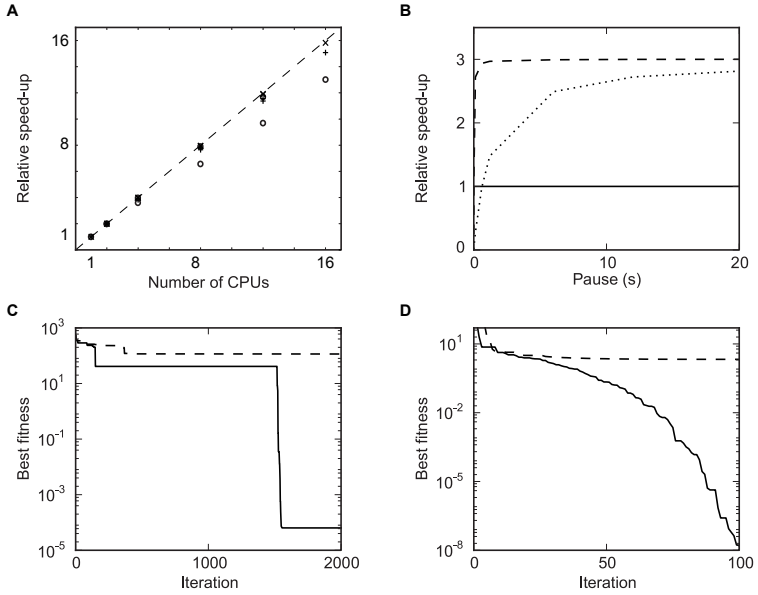

Figure 4: **A.** Speed improvements achieved when distributing independent jobs across several CPUs and computers compared to a single CPU. 48 jobs consisting of a pause (+: five seconds; ×: one second) or a neural network simulation (○: 6.3 ± 0.5 seconds across computers) are distributed on one to four computers with one to four CPUs each. The speed improvements are calculated by comparing with the total duration obtained by running all jobs in sequence on a single CPU. **B.** Speed improvement achieved when distributing an optimisation of a fitness function with a variable pause (x-axis), which corresponds to the duration of each iteration. Dashed line: GA, dotted line: PSO, solid line: single CPU. **C.** Convergence comparison on the Schwefel function between the GA without island (dashed line) and the GA with 4 islands (solid line). The y-axis represents the best fitness obtained up to the current iteration. The final best fitness is expected to be 0. **D.** Convergence comparison on the Rosenbrock function between the PSO (dashed line) and the CMA-ES (solid line).

are shown in Figure 4C, illustrate the fact that the island topology of the GA, which allows different populations to grow independently, prevent the algorithm from getting stuck in local minima. This escape from a local minima to the global one introduces a jump in the fitness convergence, as can be seen for the GA with islands but not for the one without islands. Therefore, as well as speeding up the computation, distributing certain optimisation algorithms can improve their convergence properties.

Secondly, we want to study the overhead due to the parallelization against the complexity of the fitness. If we assume that the time to compute the fitness of a population scales linearly with its size, the computation time spent for one iteration is #particles/#workers $* t_{\text{fitness}} + t_{\text{communication}} + t_{\text{update}}$. When t_{fitness}, the time needed to compute the fitness of one particle, is very small, the communication time will be bigger than the time gained by parallelization. If t_{fitness}, and therefore its complexity, is very high, there will be a significant speed improvement with parallelization. This can be seen in Figure 4B where the case of one worker on one machine (solid line) is compared with the one with 3 workers on 3 servers. In this example we model the effects of fitness functions of varying complexity by introducing a pause in the fitness evaluation for each particle, where the duration of the pause models the complexity of the function. The results are different for the GA (dashed line) and the PSO (dotted line) because the GA requires much less communication (20 times less here, as the migration from island to island is done every 20 iterations). In this case, the ideal speed improvement of 3 times is achieved almost immediately while for the PSO the fitness computation needs to be more complex to achieve the same improvement.

5. Related work

In this section we briefly review the existing Python packages for parallel and distributed computing, and highlight how they differ from Playdoh.

The most commonly addressed problem is the independent or "embarassingly" parallel problem, and there are a number of Python packages which allow the use of multiple CPUs on a single computer or across multiple computers.

Some of these include sophisticated load balancing features which Playdoh currently lacks. These packages include Celery[8], Disco[9], Jug[10], PaPy[11], papyros[12], Parallel Python[13] and superpy[14].

There are also several packages which allow access to high performance low-level APIs for distributed computing, including the Message Passing Interface (MPI) [11] and Parallel Virtual Machine (PVM). These include mpi4py[15], PyMPI[16], PyPar[17] and pypvm[18]. These packages allow for extremely low latency and high bandwidth communication between processes where it is available, but at the cost of being difficult to use (both to install and to write programs using them). To give an example, for the Monte Carlo computation of π shown in section 2.3, with MPI you would need to write two scripts, a master script and a worker script. The worker script would need to be copied by hand to each of the worker machines (although this could be done with `ssh` if a server was installed on the worker machines). Using mpi4py, the master script would look something like:

```
from mpi4py import MPI
import numpy, sys
comm = MPI.COMM_SELF.Spawn(sys.executable,
                           args=['cpi.py'],
                           maxprocs=5)
N = numpy.array(100, 'i')
comm.Bcast([N, MPI.INT], root=MPI.ROOT)
PI = numpy.array(0.0, 'd')
comm.Reduce(None, [PI, MPI.DOUBLE],
            op=MPI.SUM, root=MPI.ROOT)
print(PI)
comm.Disconnect()
```

The worker script would look like:

```
from mpi4py import MPI
import numpy
comm = MPI.Comm.Get_parent()
size = comm.Get_size()
rank = comm.Get_rank()
N = numpy.array(0, dtype='i')
comm.Bcast([N, MPI.INT], root=0)
h = 1.0 / N; s = 0.0
for i in range(rank, N, size):
    x = h * (i + 0.5)
    s += 4.0 / (1.0 + x**2)
PI = numpy.array(s * h, dtype='d')
comm.Reduce([PI, MPI.DOUBLE], None,
            op=MPI.SUM, root=0)
comm.Disconnect()
```

[8] http://celeryproject.org/
[9] http://discoproject.org/
[10] http://luispedro.org/software/jug
[11] http://code.google.com/p/papy/
[12] http://code.google.com/p/papyros/
[13] http://www.parallelpython.com/
[14] http://code.google.com/p/superpy/
[15] http://mpi4py.scipy.org/
[16] http://pympi.sourceforge.net/
[17] http://code.google.com/p/pypar/
[18] http://pypvm.sourceforge.net/

This approach undoubtedly allows for the best performance and the maximum flexibility, however for the loosely coupled problems that Playdoh addresses (beyond the independent problems), this level of performance is not necessary. In fact, Playdoh is able to provide fairly high bandwidth (although it will usually be used only in lab networks which are often not configured for the highest possible bandwidth), but cannot provide latencies as low as MPI/PVM.

Deserving of special attention is IPython [26], an extremely sophisticated shell for scientific computing with Python, NumPy and SciPy. It includes support for parallel and distributed computing, and is used by many computational scientists working with Python. Different ways of working are accommodated through "controller interfaces". At the moment it has two interfaces: one for exerting direct control of remote processes (e.g. execution of commands on a remote computer) and one for load-balanced independent task processing. For coupled problems, IPython integrates with MPI. There is no direct support for loosely coupled problems without resorting to MPI, however the architecture of IPython allows for new controller interfaces to be written. It is possible that Playdoh, or a simpler equivalent of it, could be included in IPython in this way.

Another approach taken by several packages is distributed namespaces. With this approach, objects are stored across multiple processes or machines, but a single namespace is available to the user. If the user requests a variable from this namespace, it is fetched from the process/machine that is storing it, and similarly for changing the values of variables. Independent and loosely coupled problems can be addressed within this framework, although there is no direct support for them. Packages which use this approach include NetWorkSpaces[19] and Pyro[20].

Finally, it is possible to use more generic packages to implement different approaches, for example using the Python multiprocessing package, low-level internet protocol packages, or a process managing package such as execnet[21]. This approach is the most general, of course, but requires users to write substantial amounts of code for basic operations.

What distinguishes Playdoh from the packages above is the emphasis on the typical scientific lab environment, the loosely-coupled problem framework and the built in optimisation package. In addition, it has very minimal prerequisites in comparison to some of the packages above (notably IPython, which for the parallel interfaces requires several large Python packages, including zope.interface, Twisted, Foolscap and pyOpenSSL).

With Playdoh, in a lab having several connected computers, each member of the group can distribute their computational tasks over the available nodes, keeping their own resources for themselves or allocating some resources for others to use, including the ability to dynamically change resource allocations without restarting. This peer approach differs from the master/slave architecture typical to the packages listed above, which is better adapted to a cluster environment (i.e. one in which there are many available machines not being operated directly by a user).

6. Discussion

Development of Playdoh is ongoing. We have several features planned for inclusion in future releases of Playdoh which will increase ease-of-use and allow for a more efficient distribution of resources over the cloud. A simple form of load balancing will be implemented so that resources are automatically split among computers, and, in the case of independent parallel problems, can be reallocated if the number of available CPUs or GPUs on a computer changes. Fault tolerance would allow one machine to die without jeopardizing the whole simulation thanks to redistribution and a rescheduling of the processes. We may also integrate Playdoh into IPython as an IPython controller, allowing the ease of use of Playdoh's framework for loosely-coupled problems together with the powerful features of IPython. Playdoh could also be used as a thin external Python layer to parallelize existing serial code. A serial computational task is decomposed into Python functions that could be wrapped up by Playdoh, providing an automatic parallelization of the original program flow [22].

The Playdoh package is one solution to the problem of distributing computations, emphasizing a particular balance between ease-of-use and functionality. Playdoh avoids many of the complications of MPI as we do not need the ultra-low latency and high bandwidth which it is designed for. Playdoh will therefore be useful for scientists in labs with computational problems more complicated than embarrassingly parallel ones, but for whom MPI is overkill, and who have an existing stock of user machines rather than a dedicated cluster.

[19] `http://www.lindaspaces.com/products/NWS_overview.html`
[20] `http://www.xs4all.nl/~irmen/pyro3/`
[21] `http://codespeak.net/execnet/`

Acknowledgements

This work was supported by the European Research Council (ERC StG 240132).

Conflict of Interest Statement

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

[1] Abramson, D., Bethwaite, B., Enticott, C., Garic, S., Peachey, T., Michailova, A., Amirriazi, S., 2010. Embedding optimization in computational science workflows. Journal of Computational Science 1 (1), 41–47.

[2] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., 2006. The landscape of parallel computing research: A view from Berkeley. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, 2006–183.

[3] Beazley, D. M., 2009. Python essential reference. Addison-Wesley Professional.

[4] Brette, R., Rudolph, M., Carnevale, T., Hines, M., Beeman, D., Bower, J. M., Diesmann, M., Morrison, A., Goodman, P. H., Harris, F. C., Zirpe, M., Natschläger, T., Pecevski, D., Ermentrout, B., Djurfeldt, M., Lansner, A., Rochel, O., Vieville, T., Muller, E., Davison, A. P., Boustani, S. E., Destexhe, A., 2007. Simulation of networks of spiking neurons: a review of tools and strategies. Journal of Computational Neuroscience 23, 349–98.

[5] Bücker, H., Fortmeier, O., Petera, M., 2011. Solving a parameter estimation problem in a three-dimensional conical tube on a parallel and distributed software infrastructure. Journal of Computational Science.

[6] Choudhary, A., Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Ranka, S., Saltz, J., 1992. Software support for irregular and loosely synchronous problems. Computing Systems in Engineering 3 (1-4), 43–52.

[7] Chrisochoides, N., Houstis, E., Rice, J., 1994. Mapping algorithms and software environment for data parallel PDE iterative solvers. Journal of Parallel and Distributed Computing 21 (1), 75–95.

[8] Goldberg, D. E., 1989. Genetic algorithms in search, optimization, and machine learning. Addison-wesley Reading Menlo Park.

[9] Goodman, D., Brette, R., 2008. Brian: A simulator for spiking neural networks in Python. Frontiers in Neuroinformatics 2, 5, PMC2605403.

[10] Goodman, D. F. M., Brette, R., 2009. The Brian simulator. Frontiers in Neuroscience 3 (2), 192–197.

[11] Gropp, W., Lusk, E., Skjellum, A., 1999. Using MPI: Portable parallel programming with the Message-Passing Interface. MIT Press.

[12] Hansen, N., Ostermeier, A., 2001. Completely derandomized self-adaptation in evolution strategies. Evolutionary Computation 9 (2), 159–195.

[13] Kennedy, J., Eberhart, R. C., 1995. Particle swarm optimization. In: Proceedings of IEEE international conference on neural networks. Vol. 4. Piscataway, NJ: IEEE, pp. 1942–1948.

[14] Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A., nov 2009. PyCUDA: GPU run-time code generation for high-performance computing. 0911.3456.

[15] Kumar, V., 2002. Introduction to Parallel Computing. Addison-Wesley Longman Publishing Co., Inc.

[16] Leist, A., Playne, D., Hawick, K., 2010. Interactive visualisation of spins and clusters in regular and small-world Ising models with CUDA on GPUs. Journal of Computational Science 1 (1), 33–40.

[17] Lim, D., Ong, Y., Jin, Y., Sendhoff, B., Lee, B., may 2007. Efficient hierarchical parallel genetic algorithms using grid computing. Future Generation Computer Systems 23 (4), 658–670.

[18] McInnes, L., Allan, B., Armstrong, R., Benson, S., Bernholdt, D., Dahlgren, T., Diachin, L., Krishnan, M., Kohl, J., Larson, J., 2006. Parallel PDE-based simulations using the common component architecture. Numerical Solution of Partial Differential Equations on Parallel Computers, 327–381.

[19] Morton, K. W., Mayers, D. F., 2005. Numerical solution of partial differential equations: an introduction. Cambridge Univ Pr.

[20] Mueller, C. L., Baumgartner, B., Ofenbeck, G., Schrader, B., Sbalzarini, I. F., 2009. pCMALib: a parallel Fortran 90 library for the evolution strategy with covariance matrix adaptation. In: Proceedings of the 11th Annual conference on Genetic and evolutionary computation. ACM, pp. 1411–1418.

[21] Nakasato, N., 2011. Implementation of a parallel tree method on a GPU. Journal of Computational Science.

[22] Nilsen, J., Cai, X., Høyland, B., Langtangen, H., 2010. Simplifying the parallelization of scientific codes by a function-centric approach in Python. Computational Science & Discovery 3, 015003.

[23] Oliphant, T. E., 2007. Python for scientific computing. Computing in Science & Engineering, 10–20.

[24] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krger, J., Lefohn, A. E., Purcell, T. J., 2007. A survey of general purpose computation on graphics hardware. In: Computer graphics forum. Vol. 26. Wiley Online Library, pp. 80–113.

[25] Paszynski, M., Pardo, D., Paszynska, A., 2010. Parallel multi-frontal solver for p adaptive finite element modeling of multi-physics computational problems. Journal of Computational Science 1 (1), 48–54.

[26] Perez, F., Granger, B. E., 2007. IPython: a system for interactive scientific computing. Computing in Science & Engineering, 21–29.

[27] Rossant, C., Goodman, D. F. M., Platkiewicz, J., Brette, R., 2010. Automatic fitting of spiking neuron models to electrophysiological recordings. Frontiers in Neuroinformatics 4.

[28] Rossant, C., Goodman, D. F. M., Fontaine, B., Platkiewicz, J., Brette, R., 2011. Fitting neuron models to spike trains. Frontiers in Neuroscience 5.

[29] Schutte, J. F., Reinbolt, J. A., Fregly, B. J., Haftka, R. T., George, A. D., 2004. Parallel global optimization with the particle swarm algorithm. International Journal for Numerical Methods in Engineering 61 (13), 2296–2315.
[30] Vecchiola, C., Pandey, S., Buyya, R., 2009. High-performance cloud computing: A view of scientific applications. In: 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks. IEEE, pp. 4–16.
[31] Whitley, D., Rana, S., Heckendorn, R., 1999. The island model genetic algorithm: On separability, population size and convergence. CIT. Journal of computing and information technology 7 (1), 33–47.