Modules to teach parallel and distributed computing using MPI for Python and Disco

José Ortiz-Ubarri
Computer Science Department
University of Puerto Rico - Río Piedras
Email: jose.ortiz23@upr.edu

Rafael Arce-Nazario
Computer Science Department
University of Puerto Rico - Río Piedras
Email: rafael.arce@upr.edu

Edusmildo Orozco
Computer Science Department
University of Puerto Rico - Río Piedras
Email: edusmildo.orozco@upr.edu

***Abstract*—The ability to design effective solutions using parallel processing should be a required competency for every computing student. However, teaching parallel concepts is sometimes challenging and costly, specially at early stages of a computer science degree. For such reasons we present a set of modules to teach parallel computing paradigms using as examples problems that are computationally intensive, but easy to understand and can be easily implemented using the Python parallelization libraries MPI for Python and Disco.**

***Keywords*—parallel computing, mpi, mapreduce, master worker**

I. Introduction

Hands-on programming exercises and code demonstrations/experiments are an essential educational resource for teaching introductory parallel and distributed computing (PDC) concepts. The design of effective and applicable exercises can be challenging to the instructor, especially since many libraries or frameworks for parallel programming require intricate system setups and training students in low-level details to get even the most basic programs to work. In our teaching experience, low-level details in parallel computation exercises lead, among other things, to student (and instructor) frustration and deter the students from focusing on the essential concepts [13].

Most of the available resources to teach parallelism using Python concentrate in multiprocessing programming [16]. Python also has a nice multi-threading library that can be used to introduce PDC concepts. However it does not provide multicore scalability [2]. The multiprocessing library also introduces the concept of inter-process communication through message passing. Nonetheless, using the multiprocessing library requires students to learn a new set of inter-process communication constructs instead of using established standards such as the message passing interface (MPI). Furthermore, nowadays the Map reduce model is widely used in industry and academia to solve big data problems [8].

We believe that the MPI for Python [6] library and the Disco framework [12] are simple enough to introduce students PDC concepts while giving them an experience close to what is used in real-world high performance computing applications. We also have found that bypassing the need to implement locks in the modules is an advantage especially when the modules are used in early CS core courses.

We have designed a set of easy-to-deploy hands-on educational modules that can be deployed in multi-core computers, computer clusters, and specially in the LittleFe [15] educational Cluster [14]. The educational modules can be used to introduce students PDC concepts such as: the map reduce and master-slave models, load distribution, fault tolerance, scaling, and security and privacy. The programming exercises use the Python MPI for Python and Disco (map-reduce) libraries to deemphasize attention to low-level details and facilitate solutions to more elaborate problems. Each module is constructed around a real-world or research problem to showcase the importance of parallel computing in practice and to motivate the students. An extended abstract about the modules was presented in [14]. Here we extend the work and share our experiences deploying them in our CS department.

Our modules include:

- One module with instructions to install MPI for Python and Disco to the Little Fe cluster.
- Modules to introduce MPI and MapReduce.
- Hands On modules with computational problems to be solved in parallel with MPI or MapReduce.

For those interested in testing/using our modules, source code can be found at [1].

II. Why Python?

Currently, the most popular introductory teaching language at top US universities is Python [9]. In our computer science department, C++ is the main language for the introduction to programming and data structure courses, but most of the advanced courses use Python as their main language for assignments and projects.

The MPI for Python library provides bindings to the MPI standard and support for communications of general Python objects including communication of numeric arrays which significantly ease the passing of data from one process to another [7]. For a quick comparison, Figure 6 has side by side implementations of the classic MPI Hello World program in C++ and Python.

What the program does is very simple. When the program is executed num_procs is the number of processes created that will execute the same code. Each process gets an unique rank from 0 to num_procs - 1. If the process has the rank 0, it displays the message and then receives the message from each other process and displays it when it is received. All the other

Python

```python
stat = MPI.Status()

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
num_procs = comm.Get_size()

msg = "Hello world, proc %s !" % rank

if rank == 0:    # Master work
  print msg
  for i in range(num_procs - 1):
    msg = comm.recv(source=MPI.ANY_SOURCE,
                    tag=MPI.ANY_TAG,
                    status=stat)
  print msg


else:            # Workers work
 comm.send(msg, dest = 0)
```

C++

```cpp
char msg[max_msg_length+1];
MPI_Status status ;
int rank, num_procs, tag, mpi_error ;

mpi_error = MPI_Init(&argc, &argv);
mpi_error = MPI_Comm_rank(MPI_COMM_WORLD,&rank);
mpi_error = MPI_Comm_size(MPI_COMM_WORLD,&num_procs);

msg = sprintf(message, "Hello world, proc %d!",rank);

if (rank == 0) {  // Master work
  printf(message) ;
  for (int i = 0; i < num_procs-1; i++){
    mpi_error = MPI_Recv(msg, max_msg_length+1,MPI_CHAR,
                         MPI_ANY_SOURCE, tag,
                         MPI_COMM_WORLD,&status);
    printf(msg);
  }
}
else              // Workers work
  mpi_error = MPI_Send(msg, strlen(msg)+1, MPI_CHAR,
                       0, tag, MPI_COMM_WORLD);

mpi_error = MPI_Finalize();
```

Fig. 1. Hello world listings in Python and C++ MPI.

processes from rank 1 to rank num_procs - 1 just send the message to the process with rank 0.

From the code it is easy to see that for each function in Python there is a matching C++ version. Notice that in the Python version the data passing part is simplified by not having to specify a buffer size in the send and receive function. In MPI for Python you could replace the string message by any general Python object without further MPI specifications contrary to the case of C++ where to send an object, you will need to know the memory layout and then build an special MPI data-type.

Disco is a lightweight, open-source framework for distributed computing based on the MapReduce paradigm. Disco is very powerful and easy, thanks to the Python platform. It is similar to the more popular Hadoop framework, and is also used for the distributed processing of large data sets across clusters of computers.

III. The Modules

In this section we will present some of the modules developed to introduce parallel computing in our CS curriculum and workshops for undergraduate students.The topics used in each of the modules have shown to be engaging for students. Demonstrations of the power of parallel computing to solve real-world problems could be done in any introductory CS course. For instance the password cracking example of section III-B was used to illustrate the power of parallel computing in our Introduction to Cybersecurity course, which is taken by students with possibly no prior programming experience. The other modules presented in this section where implemented in a workshop. All the modules include serial and parallel solutions to the problem. We decided to do so, in order to increase the level of confidence of both instructors and students. Table III shows the potential use of our modules in the courses of a CS curriculum. Each course is matched with the learning level (using Blooms [3] classification) that the student should achieve.

TABLE I. Potential use of our modules in CS courses and their Bloom cognitive levels (K, C, A). K: Know the term, C: Comprehend so as to paraphrase/illustrate, A:Apply it in some way.

Module	Topics	CS0	CS1	CS2	Adv
Master / Worker Costas arrays	Concurrency	C	C	A	A
	Tasks and Threads	K	C	A	A
	Performance metrics	K	C	A	A
	Divide and conquer	K	K	C	A
	Message Passing	K	K	A	A
	Reduction	K	K	A	A
Div & Conq password cracking	Concurrency	C	C	A	A
	Tasks and Threads	K	C	A	A
	Performance metrics	K	C	A	A
	Divide and conquer	K	K	C	A
	Message Passing	K	K	A	A
	Reduction	K	K	A	A
Map reduce network traffic	Concurrency	C	C	A	A
	Tasks and Threads	K	C	A	A
	Performance metrics	K	C	A	A
	Divide and conquer	K	K	C	A
	Message Passing	K	K	A	A
	Reduction	A	A	A	A

As with any other educational material related to a real-world application, the instructor should introduce the topic with simple examples, appropriate notation, and adequate data

representations.

A. Master Slave Example using MPI for Python

In this section we will describe the parallel Master / Worker implementation of the Costas arrays enumeration problem [4]. Costas arrays are known for their applications to communications and digital watermarking. The Costas array enumeration problem consists of finding permutations, of a given size N, that meet the *distinct difference property*: for all integers h, i, and j, with $1 \leq h \leq n-1$, and $1 \leq i, j \leq n-h$, $f(i+h)f(i) = f(j+h)f(j)$ implies $i = j$. Figures 2 and 3 show examples of permutations and analysis of Costas array compliance.

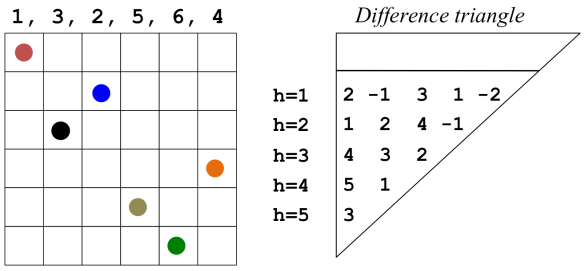

Fig. 2. A Costas array, its matrix representation, and difference triangle, which is used to check the distinct difference property. Observe that each row of the difference triangle does not contain repeated numbers.

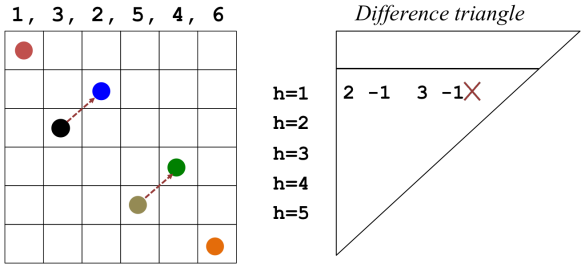

Fig. 3. A permutation that is not a Costas array. Observe that we can abort checking the distinct difference property as soon as a difference is repeated in any h.

A naïve approach to find all the Costas arrays of size N is to generate all $N!$ permutations. This search space can greatly be reduced by using a backtracking algorithm that continues searching a computational branch as long as it maintains the Costas property or stop searching in the computational branch otherwise (See Figure 4). However even this approach grows factorially with N, thus the enumeration for sizes as small as 14 may require significant execution time, e.g. in the order of hours.

We use the Costas array enumeration problem to introduce the Master / Worker paradigm using MPI for Python. As illustrated in Figure 5, the master is in charge of generating sub-permutations up to a certain length, then the sub-permutations are distributed to be completed independently by the workers. When the workers/slaves complete the search for Costas arrays of size N that start with the sub-permutation sent by the master, the results are sent back to the master.

The code in Figure 6 presents a simple and generic template of the master / worker model. The master ($rank = 0$) initially

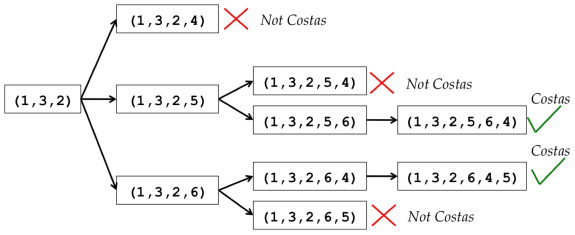

Fig. 4. Computational branch of Costas arrays of size 6 that start with sub permutation (1,3,2).

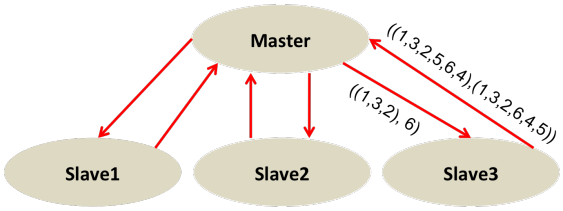

Fig. 5. Master/worker diagram of Costas problem. Master sends sub-problems to slaves. Slaves return the solution to the sub-problems.

sends work to the workers, then while there is still work to do, it will retrieve results from the workers and sends more work to them. Finally when there is no more work to do, the master collects the remaining results, and while it collects the results, it notifies the workers to stop (TAG_END). What to do with the results will depend on the problem. For instance the problem might require to store the results in a file, or that some mathematical operation is performed to obtain a final result.

The workers ($rank > 0$) will loop waiting for work from the master, then perform the work and send back the results until there is no more work to do.

The functions $getWork()$ and $handleWork()$ are what essentially distinguish the master-worker implementation for a particular problem. Function $getWork()$ should return the input that will be sent to the workers. Function $handleWork()$ should perform the work over the input and generate the result that will be sent back to the master. In the case of the Costas problem illustrated in Figure 5, the $getWork()$ function should return a sub-Costas array of size 3 (while there are arrays to be processed), and the function $handleWork()$ must generate all the Costas arrays of size 6 (if any) that start with the sub-Costas array received.

In our experience the Costas array enumeration problem can be completed during a class period of 90 minutes if the instructor provides the Master / Worker generic template and the algorithm to generate permutations (which requires the longest programming effort). The same problem can be used as an assignment by having students come up with the permutation algorithms and code for parallel distribution from scratch. Another related and fairly straightforward assignment is to generate sequences that solve the n–Queens problem [5].

B. Divide and Conquer Example using MPI for Python

Password cracking is a method by which an intruder that gains access to an encrypted passwords file can reveal the original (clear text) passwords using a dictionary and brute

```
comm = MPI.COMM_WORLD
num_procs = comm.Get_size()
rank = comm.Get_rank()
stat = MPI.Status()

if rank == 0:    # Master work
 # Send initial work
 for i in range(1, num_procs):
  worker_data = getWork()
  comm.send(worker_data, dest=i, tag=TAG_WORK)

 # Loop while there is work
 while worker_data = getWork():
  result = comm.recv(source=MPI.ANY_SOURCE,
                     tag=MPI.ANY_TAG, status=stat)
  comm.send(worker_data, dest=stat.Get_source(),tag=TAG_WORK)

 # Pick the last results and tell workers to stop
 for i in range(1, num_procs - 1):
  result = comm.recv(source=MPI.ANY_SOURCE,
                     tag=MPI.ANY_TAG, status=stat)
  comm.send("", dest=stat.Get_source(), tag=TAG_END)

else:             # Worker work
 work = comm.recv(source=0, tag=MPI.ANY_TAG, status=stat)

 while stat.Get_tag() == TAG_WORK:
  result = handleWork(work)
  comm.send(result, dest=0)
  work = comm.recv(source=0, tag=MPI.ANY_TAG, status=stat)
```

Fig. 6. Generic Python template for the Master / Worker model

force attack. Each word in a dictionary is encrypted and compared to the encrypted passwords in the password file. In our opinion, the password cracking problem combines a set of desired properties in an exercise to illustrate parallel computation. Firstly, it relates to cybersecurity - currently hot area of interest in our students. Secondly, it is easy to explain. Thus, even when a student does not fully understand the parallel concepts, hes been shown how easy it is to recover weak passwords.

It has been found that the use of real-world problems to teach introduction to programming to undergraduates results in better retention and engagement of minorities in computing related fields [10], [11]. In our experience, we also have observed that applications and programming problems related to Cybersecurity are engaging and motivating for a wide group of students.

We use the password cracking example to teach how to solve problems by dividing the problem in sub-problems of approximately the same size among the available computing nodes. In our simple approach the password file is sent to all the compute nodes (MPI broadcast), and the dictionary file is divided (approximately) equal to the compute nodes. Notice that we could have opted to broadcast both the password file and the dictionary file. Our decision to broadcast only the password file promotes a discussion in the classroom about file sizes and communication delays. The instructor can justify the implementation decision with several arguments: (1) the password file size is much smaller than the dictionary file, (2) the dictionary file is commonly installed with the operating systems of the compute nodes while the password file is not.

We have deployed this module in workshops and in the

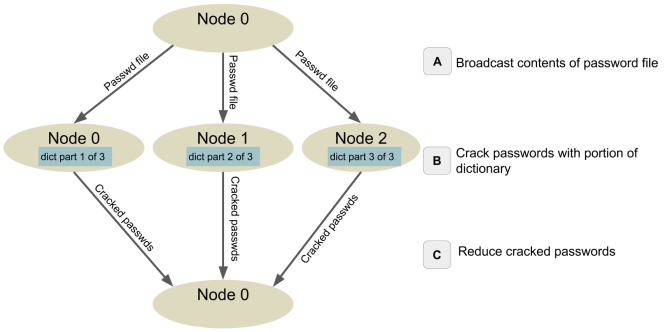

Fig. 7. Map/Reduce diagram of the password cracking problem. Node 0 broadcasts the password file to all nodes. The dictionary is divided in approximately the same size for each node. Cracked passwords are sent back to the master.

classroom, and the topic has resulted to be very engaging to the students. In very early introductory courses this module has been used to show the students the scalability achieved by dividing the problem in sub-problems and using parallel computing. In more advanced courses the students are asked to implement the parallel algorithm, after being introduced to MPI.

The main struggle that our students exhibit when facing this kind of problem is dividing an input of size M equally among N nodes when M is not a multiple of N. They either fail to recognize that they have left part of the problem unsolved, e.g. distribute $\lfloor \frac{M}{N} \rfloor$ jobs to every node, or choose to assign the remaining jobs to one of the nodes, e.g. assign $\lfloor \frac{M}{N} \rfloor + (M \bmod N)$ to the last node. The latter option is less harmful than the former. However the execution time of the program will be bounded by the node with the unbalanced load while the rest of the nodes may be idle for a significant part of the time. We ask the students to reflect on the problem, followed by a discussion of the code in Listing 1.

Listing 1. Given a problem size, the number of processes and the rank of the processes compute the portion of the problem for the rank

```
def computeIndices(size, nprocs, rank):
 portion = size  / nprocs
 residue = size % nprocs

 if rank <= residue:
  start = rank * (portion+1)
 else:
  start = rank * portion + residue

 finish = start + portion
 if rank < residue:
  finish += 1

 return start, finish
```

C. Map/Reduce Example using Disco

The aggregation of traffic data per host helps to identify anomalies in a computer network like: (1) an IP in use that is not delegated, (2) an IP generating more traffic than normal, (3) an IP that is not generating traffic.

NetFlow is a network protocol that aggregates connection information from one host to another over a certain period

of time (flow). NetFlow files consist of large quantities of connection information such as IP addresses of the source and destination hosts, and the traffic in each connection (e.g., 5 minutes of UPR network traffic can be as high as 6.5MB, or 363,149 lines of flows).

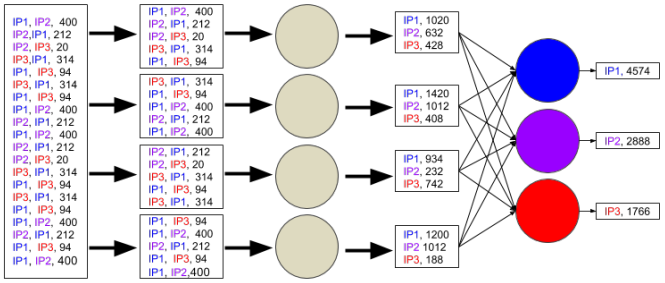

Fig. 8. Map/Reduce diagram of the NetFlow data processing example.

In this module, we use NetFlow data processing as an example of an application that can be accelerated using Disco MapReduce. Simplified code for this application is presented in Listings 2 and 3, where in (2) the source and destination IPs are mapped by their addresses and valued by their traffic octets (3) the reducer gathers the data, sorts the data by keyvalues, and computes their aggregated sums. The results generated by this simple code is a list of IP addresses with the amount of data that has been transferred in the Network2.

Listing 2. Map function using Disco

```
def map(line, params):
 #Split the data into an array
 data = line.split()
 yield data["sIPAddr"], int(data["Octets"])
 yield data["dIPAddr"], int(data["Octets"])
```

Listing 3. Reduce function using Disco

```
def reduce(line, params):
 for ip, traffic in kvgroup(sorted(iter)):
  yield ip, sum(traffic)
```

Listing 4. Main function using Disco to print the IP address in the network with the amount of traffic they have generated.

```
if __name__ == '__main__':
 job = Job().run(input=["file:///netflow.txt"],
  map=map, reduce=reduce, params=None)

 for ip, traffic in result_iterator(job.wait(
   show=True)):
  print(ip, traffic)
```

Disco provides such a high-level interface to the Map/Reduce environment that the network flow example can even be used in introductory programming courses. Its performance can be compared to a serial implementation using dictionaries in Python. In upper-level courses the example is also useful to exemplify the trade-offs between master/worker vs. Map/Reduce, e.g. the improved speedup in MPI is only achieved after the effort of hand-coding a good distribution of work to the nodes.

IV. Conclusion

In this work we present a set of modules that can be used to introduce PDC concepts in the CS curriculum using standard libraries widely used in high performance computing applications. The modules use real-world examples to engage students in the hands-on experience. We also present PDC concepts that can be introduced per course of the CS curriculum.

The presented modules are easy to set up by the instructor even in low-cost parallel platforms such as the LittleFe. Their focus on real-world examples makes them a great resource to provide hands-on experiences on PDC concepts to students starting at the introductory courses. The modules make use of libraries used in real high performance computing applications, setting the stage for more advanced exercises in later courses and/or industrial or academic research experiences for students.

Acknowledgment

The authors would like to thank the NSF-SFS award DUE-1245744 for supporting this work.

References

[1] Modules to teach pdc concepts - source code. https://bitbucket.org/cheojr/cpath/.

[2] threading - Higher-level threading interface, Python Standard Library. https://docs.python.org/2/library/threading.html. Accessed:2016-01-13.

[3] Lorin W Anderson, David R Krathwohl, and Benjamin Samuel Bloom. *A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives*. Allyn & Bacon, 2001.

[4] Rafael Arce-Nazario, Jose R Ortiz-Ubarri, et al. Enumeration of Costas arrays using GPUs and FPGAs. In *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pages 462–467. IEEE, 2011.

[5] D Bollman and E Orozco. A Faster Algorithm for the n-Queens Problem. *Congressus Numerantium*, pages 193–200, 2001.

[6] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. MPI for Python. *Journal of Parallel and Distributed Computing*, 65(9):1108 – 1115, 2005.

[7] Lisandro Dalcín, Rodrigo Paz, Mario Storti, and Jorge D'Elía. MPI for Python: Performance improvements and MPI-2 extensions. *Journal of Parallel and Distributed Computing*, 68(5):655 – 662, 2008.

[8] Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. Efficient big data processing in Hadoop MapReduce. *Proceedings of the VLDB Endowment*, 5(12):2014–2015, 2012.

[9] Philip Guo. Python is now the most popular introductory teaching language at top us universities. *BLOG@ CACM, July*, 2014.

[10] Lucas Layman, Laurie Williams, and Kelli Slaten. Note to self: make assignments meaningful. *ACM SIGCSE Bulletin*, 39(1):459–463, 2007.

[11] Jane Margolis and Allan Fisher. *Unlocking the clubhouse: Women in computing*. MIT press, 2003.

[12] Prashanth Mundkur, Ville Tuulos, and Jared Flatow. Disco: A computing platform for large-scale data analytics. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, Erlang '11, pages 84–89, New York, NY, USA, 2011. ACM.

[13] Edusmildo Orozco, Rafael Arce-Nazario, Jose Ortiz-Ubarri, and Humberto Ortiz-Zuazaga. A Curricular Experience With Parallel Computational Thinking: A Four Years Journey. In *Proceeding of EduPDHPC*, 2013.

[14] José Ortiz-Ubarri and Rafael Arce-Nazario. Modules to teach parallel computing using Python and the LittleFe cluster. *Super Computing*, 2013.

[15] Charles Peck. LittleFe: parallel and distributed computing education on the move. *Journal of Computing Sciences in Colleges*, 26(1):16–22, 2010.

[16] Sushil K Prasad, Anshul Gupta, Arnold L Rosenberg, Alan Sussman, and Charles C Weems. Topics in parallel and distributed computing: Introducing concurrency in undergraduate courses. 2015.