QUICKLB

Release 0.1

Victor Azizi

May 31, 2022

CONTENTS:

QUICKLB is a very efficient library that sacrifices optimality for speed. It shines in distributed applications that have unpredictable unbalanced computational peaks in their domains. To begin using QUICKLB we refer you to the *Installation* section and the example in the *Example usage* section.

API documentation can be found in the *Loadbalancer* section

CHAPTER

ONE

INSTALLATION

Quick and easy installation can be done with conda by creating an environment from the included conda_env.yml file and installing quicklb into this environment:

```
$ conda create -c conda-forge -n quicklb --file conda_env.yml
$ conda activate quicklb
$ python setup.py install
```

1.1 Manual installation

Make sure you have the folling packages available:

- cmake **>= 3.22**
- gfortran **>= 11.2** or ifort **>= 2022.1**
- openmpi **>= 4.1.2**

For python functionality:

- python **>= 3.9**
- numpy **>= 1.20**
- mpi4py **>= 3.1.1**
- matplotlib **>= 3.5.1**
- scipy **>= 1.8**

The library can be created through cmake directly, the CMakeFile honors the CONDA_PREFIX if present, and tries to install there instead of the default install location:

```
$ mkdir build
$ cd build
$ cmake ../ -DFortran_COMPILER=mpif90 -DCMAKE_BUILD_TYPE={RELEASE,DEBUG}  -DPYTHON_
↪INTERFACE={off,on}
$ make
$ make install
```

Or it can be created throught the python setuptools:

```
$ python setup.py install
```

CHAPTER

TWO

EXAMPLE USAGE

To use QUICKLB it has to be imported first, if this fails please go to the *Installation* section.

```
import quicklb
```

2.1 Unbalanced reaction diffusion with QUICKLB

In this example we will create a reaction diffusion simulation which uses a loadbalancer for the chemistry calculation part (which is local), and a normal non-loadbalanced process for the diffusion process.

The easies and quickest way to get started is with the Loadbalancer object.

```
from quicklb import Loadbalancer
```

We have to create our domain, and work arrays and divide them over the number of processors, to reduce the amount of communication code necessary for the non-loadbalanced part we use shared-memory array (which are a bit more complex than normal numpy arrays). It also means that the example will only work if all processors have acces to the same memoryspace.

```
import numpy as np
from mpi4py import MPI
import matplotlib.pyplot as plt
import scipy.integrate
comm = MPI.COMM_WORLD

def allocate_shared_array(size):
  nbytes = np.prod(size) * MPI.DOUBLE.Get_size()
  if comm.Get_rank() != 0:
    nbytes = 0
  win = MPI.Win.Allocate_shared(nbytes, MPI.DOUBLE.Get_size(),comm=comm)
  buf, _ = win.Shared_query(0)
  return np.ndarray(size,dtype=np.float64,buffer=buf), win

# Using shared array we cheat a bit to  make it simpler
size = 30
A, winA = allocate_shared_array((size,size))
B, winB = allocate_shared_array((size,size))

# Calculate which part of the shared array is "local" to this processor
remainder = size%comm.Get_size()
```

(continues on next page)

(continued from previous page)

```
slice_size = size//comm.Get_size()
start = int(slice_size*comm.Get_rank() + min(comm.Get_rank(), remainder))
end   = int(slice_size*(comm.Get_rank() + 1) + min(comm.Get_rank()+1, remainder))

A_local = A[start:end,:]
```

This gives us a slice on each processor which we fill up with an initial condition:

```

if comm.Get_rank() == 0:
  A[:,:] = 1.
  B[:,:] = 0.
  A[size//2-3:size//2+3,size//2-10:size//2+10] = 0.50
```

Secondly we need a structure to encapsulate these points such that the loadbalancer knows which data to communicate, how to encapsulate, and how to compute an interation

```

class Cell():

  def __init__(self,AA=None,BB=None):
    self.A = np.array([float('nan')],dtype=np.float64)
    self.B = np.array([float('nan')],dtype=np.float64)
    if AA != None:
      self.A = AA
    if BB != None:
      self.B = BB

  def compute(self):
    a = self.A[0]
    b = self.B[0]
    soln =scipy.integrate.solve_ivp(Cell.deriv,(0,1),(a,b),
        method='BDF', rtol=0.0001)
    self.A[0] = soln.y[0][-1]
    self.B[0] = soln.y[1][-1]

  def deriv(t,y):
    a, b = y
    abb = a*b*b
    adot =  - abb + 0.0545 * ( 1 - a )
    bdot =  + abb - ( 0.0545 + 0.062 ) * b
    return adot, bdot

  def serialize(self):
    return np.concatenate( (self.A.view(dtype=np.byte),
                            self.B.view(dtype=np.byte)))

  def deserialize(self, buffer):
    self.A[0] = buffer[0:8].view(dtype=np.float64)
    self.B[0] = buffer[8:16].view(dtype=np.float64)
```

This Cell class can either point to data when local, or store data when offloaded, furthermore a compute function is provided

We must create a list of cells that we want loadbalanced

```
cells = [Cell(A_local[i,j:j+1], B_local[i,j:j+1])
         for j in range(size) for i in range(A_local.shape[0])]
```

Then we can create the Loadbalancer:

```
lb = Loadbalancer(cells, "SORT", 0.01, 0.01, 3)
```

As an intermezzo we also need a diffusion operator which is separate from the loadbalancer:

```
A_diff = np.empty(shape=(A_local.shape[0]+2,A_local.shape[1]+2))
B_diff = np.empty(shape=(B_local.shape[0]+2,B_local.shape[1]+2))
def diffuse():
  # Perform diffusion over the local field + boundaries (this is where we cheat with␣
↪shared memory arrays
  A_diff[1:-1,1:-1] = A_local
  # local periodic boundaries
  A_diff[0,1:-1] = A[start-1,:]
  A_diff[-1,1:-1] =A[end%A.shape[0],:]
  # "communicate" boundaries,
  A_diff[1:-1,0] = A_local[:,-1]
  A_diff[1:-1,-1] =A_local[:,0]

  B_diff[1:-1,1:-1] = B_local
  # local periodic boundaries
  B_diff[0,1:-1] = B[start-1,:]
  B_diff[-1,1:-1] =B[end%B.shape[0],:]
  # "communicate" boundaries,
  B_diff[1:-1,0] = B_local[:,-1]
  B_diff[1:-1,-1] =B_local[:,0]

  LA       =                        A_diff[ :-2, 1:-1] + \
            A_diff[1:-1, :-2] - 4*A_diff[1:-1, 1:-1] + A_diff[1:-1, 2:] + \
                              +   A_diff[2:  , 1:-1]
  LB       =                        B_diff[ :-2, 1:-1] + \
            B_diff[1:-1, :-2] - 4*B_diff[1:-1, 1:-1] + B_diff[1:-1, 2:] + \
                              +   B_diff[2:  , 1:-1]
  return LA,LB
```

And with this loadbalancer + diffusion operator we can iterate to our heart's content:

```
number_of_iterations = 100
lb.partition()
for i in range(number_of_iterations):
  winA.Sync()
  winB.Sync()
  comm.Barrier()
  LA,LB = diffuse()

  # Chemistry
  if i%10 == 1:
    lb.partition()
    lb.partitioning_info()
```

(continues on next page)

(continued from previous page)

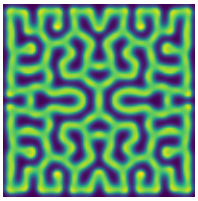
```
  if i%10 == 0:
    if comm.Get_rank() == 0:
      plt.imshow(B)
      plt.show(block=False)
      plt.pause(0.03)
  lb.iterate()

  A_local[:,:] = A_local + 0.1*LA
  B_local[:,:] = B_local + 0.05*LB
```

Lastly we can create visualise this quite easily with matplotlib!

```
winB.Sync()
comm.Barrier()
if comm.Get_rank() == 0:
  plt.imsave("gray_scott_rd.png",B)
  plt.show()
```

Timings	Normal	Load Balanced
2 MPI procs	50.31 s	51.63 s
3 MPI procs	46.02 s	41.25 s
4 MPI procs	39.54 s	33.04 s



CHAPTER

THREE

LOADBALANCER

class quicklb.**Loadbalancer**(*objects*, *algorithm*, *max_abs_li*, *max_rel_li*, *max_it*)

__init__(*objects*, *algorithm*, *max_abs_li*, *max_rel_li*, *max_it*)

Create a Loadbalancer object

Parameters

- **objects** (*list<Cell>*) – A list of objects which implement all functions specified in the mock **Cell** object.
- **algorithm** (*string*) – Specifies the loadbalancing algorithm, valid values are: - *"GREEDY"* - *"SORT"* - *"SORT2"*
- **max_abs_li** (*float*) – specify the maximum absolute load-imbalance for the partitioning algorithm. When this threshold is reached the partitioning concludes *0.0* would be no load imbalance, *1.* would be a load imbalance of 100%
- **max_rel_li** (*float*) – specify the maximum relative load-imbalance for the partitioning algorithm. Is not used in the GREEDY partitioning scheme. When this threshold is reached the partitioning concludes.
- **max_it** (*int*) – Maximum number of iterations for the loadbalancing algorithm. Set this somewhere between 1 and the number of processors used (or larger, it is limited to the maximum number of processors anyway)

Returns

A very fresh loadbalancing object !

Return type

Loadbalancer

iterate()

Perform a single iteration, with computation offloading. this eventually calls **compute** on every single cell

Return type

None

partition()

Call this function to (re)-partition the cells over the processors, it is recommended to call this once before **iterate()**

Return type

None

partitioning_info(*detailed=False*)

Writes out partitioning info to the loadbalance.info file

Parameters
detailed (*bool*) – When set to *True* write detailed information about every cell to loadbalance.info

Return type
None

3.1 Abstract Cell Class

this is a possible implementation of a cell class that needs to be fed to the loadbalancer

```
import struct

class Cell():
  data: float = 42.

  def compute(self):
    self.data = 1./data

  def serialize(self):
    return struct.pack('d',self.data)

  def deserialize(self, buffer):
    self.data = struct.unpack('d',buffer.tobytes())
```

class quicklb.**Cell**

Example Cell for the Loadbalancer class that needs some work to be done

compute()

Function which is called whenever work needs to be done, as this cell can be offloaded at the time of computation, it is recommended to only use (de)serialized data for the computation, otherwise results will be undefined

deserialize(*buffer*)

Function which deserializes the object.

Parameters
buffer (*1D numpy bytes array*) –

Return type
None

serialize()

Function which serializes the object.

Return type
Bytes-like object

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

C

D

I

L

P

S