Data Processing in Serial Oscillation Protein Crystallography: *Serial_xds and O*ptimization of Processing Time Through Parallelization

Nugzari Khalvashi-Sutter

Corning Community College

Abstract:

This report documents the development of a *serial_xds* program that is used for data processing in oscillation serial crystallography. The two objectives of this project were to write a functional program in the Python language; and to introduce parallelization– distribution of tasks across several computing nodes– to reduce the overall processing time. The program was organized in four script files with clean object-oriented design, along with almost doubled efficiency in processing time after implementing parallelization.

Introduction:

Serial crystallography is a method to obtain atomic structural information from a protein crystal by collecting small X-ray diffraction datasets from many crystal specimens. When a focused X-Ray beam interacts with a protein crystal, the incident X-rays are diffracted, producing a pattern of reflections that are recorded by a detector [1]. Information about the position and intensity of reflections is transmitted to a computer in digital form and then used to reconstruct an image of the molecules in the unit cell.

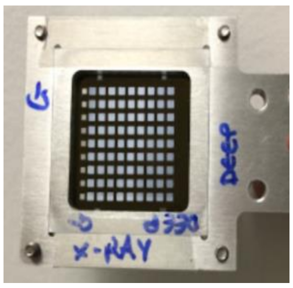


Figure 1

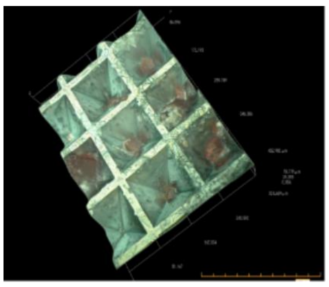


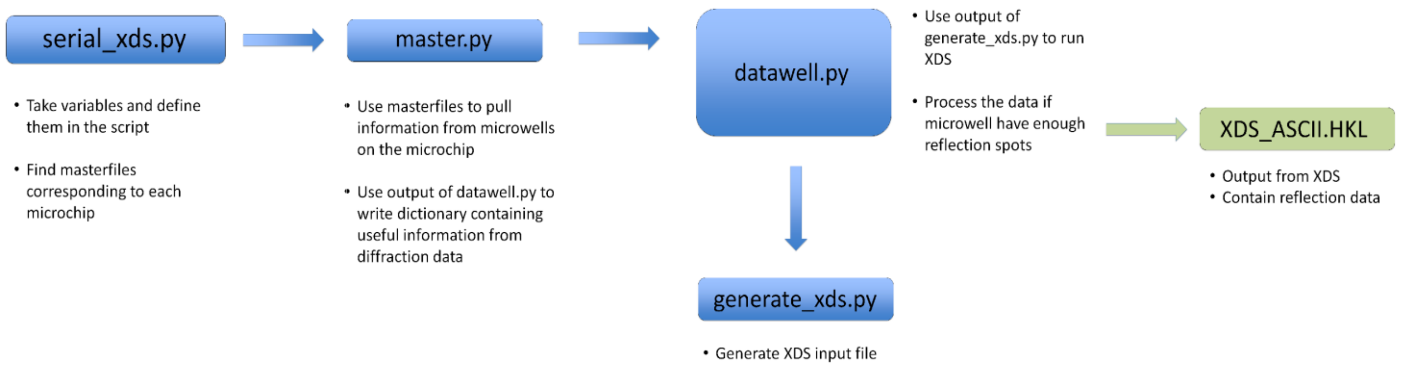
Figure 2

At the Cornell High Energy Synchrotron Source (CHESS), a microchip system is used to deliver crystals to the X-Ray beam in the serial crystallography experiment. The chip (Figure 1) contains up to 81 grids of 40x40 wells of 10-100 µm in size (Figure 2) [2]. I will be using these terms to describe some functionalities of the program in the sections below.

Program Organization:

Generally, data processing in the serial crystallography experiment involves execution of several repeating operations on the same types of data. In our case, *serial*_xds generates input files and runs X-Ray Detector Software (XDS) - a program package written for the "reduction" of 2-dimensional oscillation images obtained from irradiated crystals, developed by Wolfgang Kabsch in 1986 [3] . XDS consists of seven steps: XYCORR (detector surface and parallax correction); INIT (calculates initial background); COLSPOT (finds spots for indexing); IDXREF (indexing and refinement); DEFPIX (generates masks prior to processing); XPLAN (data collection strategy); INTEGRATE (integration of reflections); and CORRECT (applies correction factors).

Object-oriented programing (OOP) in Python, a type of computer software design in which types of a data structure are defined along with operations that can be applied to it, is an ideal solution for the structural organization. Some of the benefits of this approach include a reduced length of the code, easy debugging, and usage of the code lines. The program was organized in four scripts: *serial_xds.py, master.py, datawell.py, generate_xds.py*. The flowchart depicts the overall structure of the program:


serial_xds.py
• Take variables and define them in the script
• Find masterfiles corresponding to each microchip
master.py
• Use masterfiles to pull information from microwells on the microchip
• Use output of datawell.py to write dictionary containing useful information from diffraction data
datawell.py
• Use output of generate_xds.py to run XDS
• Process the data if microwell have enough reflection spots
generate_xds.py
• Generate XDS input file
XDS_ASCII.HKL
• Output from XDS
• Contain reflection data

serial_xds.py

- Using the *argparse* Python module to define command line options. Parsed variables include:
    - Directory or directories containing input files
    - Position (in pixels) of the beam center
    - Oscillation angle per well
    - Detector distance in mm
    - Wavelength in Ångstrom
    - Number of frames per degree
    - Output directory
    - Space group (optional)
    - Unit cell (optional)

- Searches for HDF5 master files in a given path and creates a list. Each master file corresponds to a single grid on the chip (Figure 1).
- Uses master files from the list to create instances of the Master class defined in *master.py*. Each master file is now an object.
- Uses the *time* Python module to return total processing time, after program successfully completed.

master.py

- Defines the Master class, and methods within it. Methods refer to functions that are used do something with objects passed into the class:
    - Upon initialization, creates a directory for the Master object.
    - Uses object to create instances of a class defined in *datawell.py*.
      The *Multiprocessing* Python module is used to run this part of the code in parallel. Maximum number of available cores is used by default.
    - Creates a dictionary for the object and write it as a JSON file. Dictionary store information about the outcome of XDS.

datawell.py

- Defines the Datawell class, and methods within it. (Datawell is a subclass of Master, defined in *master.py*) Methods refer to functions, that are used do something with objects passed into the class:
    - Create Datawell directory.
    - Runs *XDS* in each datawell directory.

generate_xds.py

- Generate XDS input file, using variables defined in *serial_xds.py*.

Results on parallelization:

We tested the efficiency of the program with parallelization in *serial_xds.py* and *master.py*. The best performance was observed with spreading tasks in *master.py*, reducing processing time almost by half of the time without parallelization (Table 1,2).

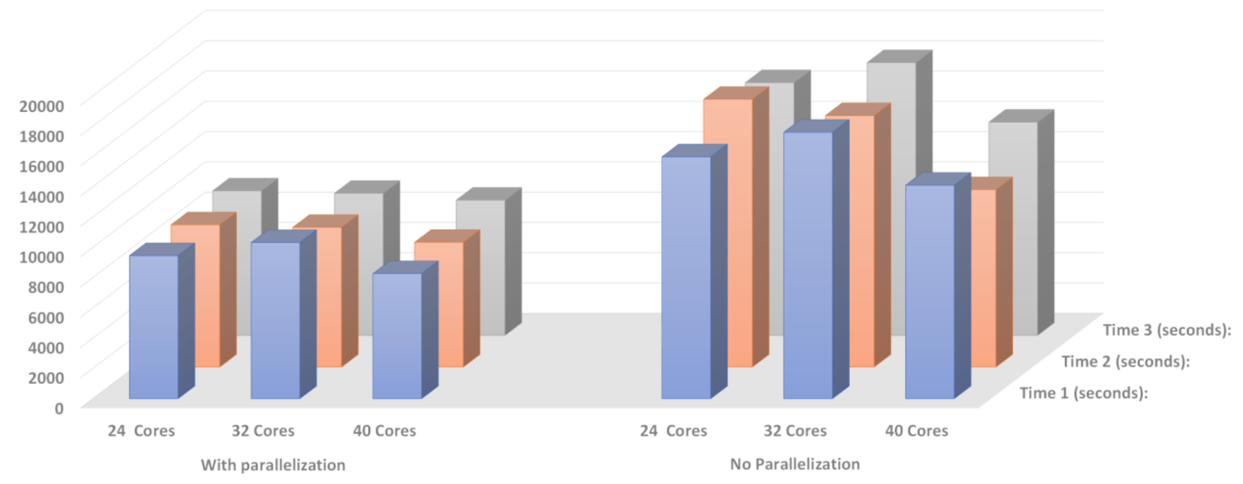


Table 1

	No Parallelization		
	24 Cores	32 Cores	40 Cores
Time 1 (seconds):	15977	17597	14099
Time 2 (seconds):	17688	16613	11718
Time 3 (seconds):	16700	18021	14089
Average:	16788	17410	13302

Table 2

	With Parallelization		
	24 Cores	32 Cores	40 Cores
Time 1 (seconds):	9445	10314	8258
Time 2 (seconds):	9396	9227	8239
Time 3 (seconds):	9552	9390	8936
Average:	9464	9644	8478

Summary:

Python made it easy to cluster parts of the data into objects and assign functionalities to them using classes. Structuring the program in this way led to reduced length of the code and an overall clean, object-oriented design. A multiprocessing package was used to leverage the availability of several processors on CHESS machines. The performance was tested by comparison of the processing time for the code with no parallelization, to the code with parallelization, showing significant improvement in processing time.

Acknowledgments:

I would like to thank my mentor Aaron D. Finke for his time, efforts, and support with this project. Summer Research for Community College Students (SERCCS) program gave me a great opportunity to work with world-class people and learn from them about their fields.

This work was done at CHESS, funded by National Institutes of Health/National Institute of General Medical Sciences award GM-103485. Experimental datafiles for testing were provided by MacCHESS ID7B2 beamline. To see more information about the program, visit: https://github.com/aaronfinke/serial_xds

References:

1. Rhodes, Gale. *Crystallography Made Crystal Clear: A Guide for Users of Macromolecular Models.* Second ed., Academic Press, 2000.

2. Wierman, J. L., et al. (2019) IUCrJ, 6, 305-316

3. Kabsch, W. (2010a). XDS. Acta Cryst. D66, 125-132

Serial_xds.py

```
import argparse, fnmatch, os, h5py, time
import master

def main():
        time1=time.time()
        parser = argparse.ArgumentParser(description='Arguments required to process the data: input, beamcenter, distance.')

        parser.add_argument('-i', '--input', type=str, nargs='+', required=True, help='Path of Directory containing HDF5 master file(s)')

        parser.add_argument('-b', '--beamcenter', type=int, nargs=2, required=True, help='Beam center in X and Y')

        parser.add_argument('-r', '--oscillations', type=float, default=1, help='Oscillation angle per well')

        parser.add_argument('-d', '--distance', type=float, required=True, help='Detector distance in mm')

        parser.add_argument('-w', '--wavelength', type=float, default=1.216, help='Wavelength in Angstrom')

        parser.add_argument('-f', '--framesperdegree', type=int, default=5, help='Number of frames per degree')

        parser.add_argument('--output', default=os.getcwd(), help='Use this option to change output directly')

        parser.add_argument('-sg', '--spacegroup', help='Space group')

        parser.add_argument('-u', '--unitcell', type=str, default="100 100 100 90 90 90", help='Unit cell')

        parser.parse_args()

        args = parser.parse_args()

        # Get all master files from the given path and create a list:
        for masterdir in args.input:
                master_list = fnmatch.filter(os.listdir(masterdir), "*master.h5")
                print(master_list)
                for masterfile in master_list:
                        # Return number of data files linked to a master file:
                        masterpath = "{}/{}".format(masterdir, masterfile)
                        totalframes = master.get_number_of_files(masterpath)

                        # Each master file in the list now used to create an instance of a class called 'Master' (from master.py):
                        master_class = master.Master(args, masterpath, totalframes)

        time2 = time.time()
        print("Total time: {:.1f} s".format(time2-time1))
if __name__=='__main__':
        main()
```

master.py

```
import os, sys, h5py, json, re
from multiprocessing import Pool
import datawell

class Master(object):

        # Generating a constructor for the class:
        def __init__(self, args, masterpath, num_of_total_frames):
                self.args = args
                self.masterpath = masterpath
                self.frames_per_degree = args.framesperdegree
                self.total_frames = num_of_total_frames
                self.output = args.output


                # Variables defined within class:
                self.master_dictionary = {}
                self.new_list = []

                # Functions called within class:
                self.create_master_directory() # creating masterfile directories

                # creating datawell directory and run XDS in it (with parallelization)
                self.new_list = map(int, range(1,self.total_frames,self.frames_per_degree))
                p = Pool()
                p.map(self.create_and_run_data_wells, self.new_list)
                p.close()



                self.generate_master_dictionary() # creating a master dictionary
                self.write_master_dictionary() # writing a master dictionary as a json file



        def create_master_directory(self):
                # Generate a name for masterfile directory:
                try:
                        end_index = self.masterpath.find('_master.h5')
                        start_index = self.masterpath.rfind('/')
                        dir_name= self.masterpath[start_index+1:end_index]
                        new_dir_path = '{new_dir}/{name}'.format(new_dir = self.output, name = dir_name)

                        # Create a mesterfile directory:
                        try:
                                os.makedirs(new_dir_path)
                        except OSError:
                                print("Creation of the directory {} failed. Such file may already exist.".format(dir_name))
                        else:
                                print("Successfully created the directory {}".format(dir_name))
                except:
                        print("Something is not working. Check the code in 'master.py'")




        def create_and_run_data_wells(self, framenum):
                # Generate datawell directories by creating instances of class called 'Datawell' (from datawell.py):
                data_well = datawell.Datawell(framenum, framenum+self.frames_per_degree-1, self.get_master_directory_path(), self.masterpath, self.args)
```

datawell.py

```
import subprocess, os, re
from generate_xds import gen_xds_text


class Datawell(object):

	# Generating a constructor for the class:
	def __init__(self, first_frame, last_frame, master_directory, masterpath, args):
		self.ff = first_frame
		self.lf = last_frame
		self.master_dir = master_directory
		self.masterpath = masterpath
		self.args = args


		# Variables defined within class:
		self.framepath = "{d}/{start}_{end}".format(d=self.master_dir, start=self.ff, end=self.lf)
		self.results_dict = {}
		self.final_dict = {}


		# Functions called within class:
		self.setup_datawell_directory() # generating datawell directory
		self.gen_XDS() # generating XDS.INP in datawell directory
		self.run()




	def setup_datawell_directory(self):
		# Generate datawell directory:
		try:
			os.makedirs(self.framepath)
		except OSError:
			print("Failed to create datawell directory")



	def gen_XDS(self):
		# Generating XDS file in datawell directory:
		try:
			d_b_s_range = "{a} {b}".format(a=self.ff, b=self.lf)
			with open(os.path.join(self.framepath, 'XDS.INP'), 'x') as input:
				input.write(gen_xds_text(self.args.unitcell, self.masterpath.replace("master", "??????"),
				self.args.beamcenter[0], self.args.beamcenter[1], self.args.distance, self.args.oscillations,
				self.args.wavelength, d_b_s_range, d_b_s_range, d_b_s_range))
		except:
			print("IO ERROR")



	def run(self):
		# Run XDS in the datawell derectory:
		os.chdir(self.framepath)
		subprocess.call(r"xds_par")
		os.chdir(self.master_dir)
```

generate_xds.py

```
def gen_xds_text(UNIT_CELL_CONSTANTS, NAME_TEMPLATE_OF_DATA_FRAMES, ORGX, ORGY, DETECTOR_DISTANCE, OSCILLATION_RANGE, X_RAY_WAVELENGTH, DATA_RANGE,
BACKGROUND_RANGE, SPOT_RANGE):
        text = """
SPACE_GROUP_NUMBER=0
UNIT_CELL_CONSTANTS= {in_1}
NAME_TEMPLATE_OF_DATA_FRAMES= {in_2}
JOB= XYCORR INIT COLSPOT IDXREF DEFPIX INTEGRATE CORRECT
ORGX= {in_3}  ORGY= {in_4}
DETECTOR_DISTANCE= {in_5}
OSCILLATION_RANGE= {in_6}
X-RAY_WAVELENGTH= {in_7}
DATA_RANGE= {in_8}
BACKGROUND_RANGE= {in_9}
SPOT_RANGE= {in_10}
DETECTOR=EIGER
MINIMUM_VALID_PIXEL_VALUE=0
OVERLOAD= 1048500
SENSOR_THICKNESS=0.32
QX=0.075  QY=0.075
NX= 1030  NY= 1065
UNTRUSTED_RECTANGLE=    0 1031    514  552
LIB=/nfs/chess/sw/macchess/dectris-neggia-centos6.so
TRUSTED_REGION=0.0 1.41
DIRECTION_OF_DETECTOR_X-AXIS= 1.0 0.0 0.0
DIRECTION_OF_DETECTOR_Y-AXIS= 0.0 1.0 0.0
MAXIMUM_NUMBER_OF_JOBS=4
MAXIMUM_NUMBER_OF_PROCESSORS=8
ROTATION_AXIS= 0.0 -1.0 0.0
INCIDENT_BEAM_DIRECTION=0.0 0.0 1.0
FRACTION_OF_POLARIZATION=0.99
POLARIZATION_PLANE_NORMAL= 0.0 1.0 0.0
REFINE(IDXREF)=BEAM AXIS ORIENTATION CELL  ! POSITION
REFINE(INTEGRATE)= ! ORIENTATION POSITION BEAM CELL AXIS
REFINE(CORRECT)=POSITION BEAM ORIENTATION CELL AXIS
VALUE_RANGE_FOR_TRUSTED_DETECTOR_PIXELS= 6000 30000
! INCLUDE_RESOLUTION_RANGE=50 1.8
MINIMUM_I/SIGMA=50.0
CORRECTIONS= !
SEPMIN=4.0
CLUSTER_RADIUS=2
        """.format(in_1=UNIT_CELL_CONSTANTS, in_2=NAME_TEMPLATE_OF_DATA_FRAMES, in_3=ORGX, in_4=ORGY,
        in_5=DETECTOR_DISTANCE, in_6=OSCILLATION_RANGE, in_7=X_RAY_WAVELENGTH, in_8=DATA_RANGE, in_9=BACKGROUND_RANGE,
        in_10=SPOT_RANGE)
        return text
```