IDNM 680: HOMEWORK 2

JAMES DELLA-GIUSTINA

Task 1

This task should be done by each group. Members in each group should work together to build a Linux Beowulf Cluster. The main purpose of this assignment is to understand the hardware of a mini Linux Cluster. A small Beowulf Linux Cluster consists of master node and a slave node. They connect through a network switch or router.

Solution 1. Using Ahmed's (wonderful) guide, we collectively worked through the steps.

```
sudo apt install openssh-server
```

For node1 (manager):

```
sudo vim /etc/hosts
```

Where we added the workers IP address.

```
ssh-keygen
ssh-copy-id -i ~/.ssh/id_rsa.pub worker
ssh worker

sudo apt install vim
sudo vim /etc/hosts # Added workers ip adress
```

On node1:

```
sudo apt install libopenmpi-dev
ssh worker
sudo apt install libopenmpi-dev
exit
mpirun -np 2 -host localhost, worker hostname
vim hosts
localhost slots = 4
worker slots = 4
mpirun -np 8 -hostfile ~/hosts hostname
sudo apt-get install nfs-kernel-server
mkdir cloud
sudo nano /etc/exports
```

Starting in the Manager node:
Then adding the following line:

```
/home/mpiuser/cloud:(rw,sync,no_root_squash,no_subtree_check)
```

Now returning to the BASH shell:

```
sudo exportfs -a
sudo service nfs-kernel-server restart
```

Date: Spring 2023.

After restarting, ssh into node 2 and execute:

```
sudo apt-get install nfs-common
cd
mkdir cloud
sudo mount -t nfts manager:/home/mpiuser/cloud ~cloud
df -h
```

After these steps, we were successfully able to compile and run a sample MPI program.

Task 2

This part should be done by each member of the group on the cluster they have built. Please complete the following subtasks: 1. Open an account for yourself. 2. Compilation and installation of BLAS, LAPACK, and SCALAPACK libraries under your home directory. 3. Installation of python, and Quantum Espresso package under your home directory.

Solution 2. (1) Using the GUI, I went into system settings and added an account named jdella and set my password.
(2) For these three libraries, I used `mkdir *software_name*` in my home directory. For BLAS and LAPACK, I used the following `BASH` commands, which were modified for the software in question.

```
wget *https://natlib/SW.tgz*
tar -xvf SW.tgz
cd SW/
make
mv SW_LINUX.a libSW.a
mv *.a /usr/local/lib
```

For SCALAPACK:

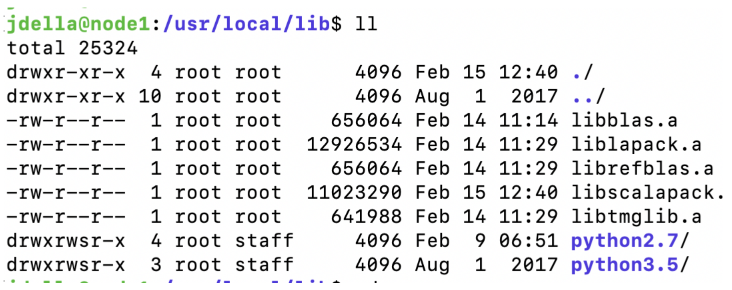
```
wget https://github.com/Reference-ScaLAPACK/scalapack/archive/refs/tags/v2.2.0.tar.gz
tar -xf v2.2.0.tar.gz scalapack-2.2.0/
vi SLmake.inc.example # Added BLAS and LAPACK library paths
make
```

For Quantum Espressoro, I used the following commands.

```
wget https://www.quantum-espresso.org/rdm-download/488/v7-1/6
   dba8073283c28fdbd784bcde9fe6e1e/qe-7.1-ReleasePack.tar.gz
tar -xf qe-7.1-ReleasePack.tar.gz qe-7.1/
cd qe-7.1/
sudo ./configure --enable-parallel=yes --enable-openmp=yes --enable-threads
make all
vi ~/.bashrc
```

I also needed to install Wannier90 to successfully run Quantum Espresso:

```
cd ~
ll
mkdir WAN90
cd WAN90/
wget https://github.com/wannier-developers/wannier90/archive/v3.1.0.tar.gz
tar -xzvf v3.1.0.tar.gz wannier90-3.1.0/
cp wannier90-3.1.0/config/make.inc.gfort ./make.inc
cd wannier90-3.1.0/
make
```



```
jdella@node1:/usr/local/lib$ ll
total 25324
drwxr-xr-x  4 root root      4096 Feb 15 12:40 ./
drwxr-xr-x 10 root root      4096 Aug  1  2017 ../
-rw-r--r--  1 root root    656064 Feb 14 11:14 libblas.a
-rw-r--r--  1 root root  12926534 Feb 14 11:29 liblapack.a
-rw-r--r--  1 root root    656064 Feb 14 11:29 librefblas.a
-rw-r--r--  1 root root  11023290 Feb 15 12:40 libscalapack.
-rw-r--r--  1 root root    641988 Feb 14 11:29 libtmglib.a
drwxrwsr-x  4 root staff     4096 Feb  9 06:51 python2.7/
drwxrwsr-x  3 root staff     4096 Aug  1  2017 python3.5/
```

Task 3

In this task, use Fortran 90 or python languages, write a parallel program to calculate the exact value of π. The accurate digit of π should be at least 100 digits after the decimal point. Please provide a performance data (using python to plot out below): total CPU time vs. Number of CPUs.

Solution 3. Claim: $\int_0^1 \frac{1}{1+x^2}\,dx = \frac{\pi}{4}$.

Proof. By applying the substitution $x = \tan\theta$. Then $dx = \sec^2\theta\,d\theta$ and the integral becomes:

$$\begin{aligned}
\int_0^1 \frac{1}{1+x^2}\,dx &= \int_0^{\pi/4} \frac{\sec^2\theta}{1+\tan^2\theta}\,d\theta \\
&= \int_0^{\pi/4} \cos^2\theta\,d\theta \\
&= \frac{1}{2}\theta + \frac{1}{4}\sin(2\theta)\Big|_0^{\pi/4} \\
&= \frac{\pi}{4}
\end{aligned}$$

□

The value of the definite integral $4\int_0^1 \frac{1}{1+x^2}\,dx$ is exactly π. Therefore, we can write a program that uses the trapezoidal rule to approximate the area under the curve of $\dfrac{1}{1+x^2}$ on the interval $[0,1]$. The program will output the value of the integral, which will be an approximation of π. Consider the function $\dfrac{1}{1+x^2}$ bounded below by the $x-$ axis, and on the sides by the vertical lines defined by $x = 0, 1$. Then, subdivide the x-axis from $[0,1]$ into any number of subdivisions n. We draw trapezoids using left and right subdivision points as the base of each trapezoid. For instance, let x_i and x_{i-1} be the right and left end points (respectively) of the i^{th} subdivision. Then, the base of the i^{th} trapezoid is given by the length $h = x_i - x_{i-1}$. Likewise, the height of the trapezoid at the left and right endpoints is given by $f(x_{i-1})$ and $f(x_i)$ respectively, as shown by Figure 1. The area of this trapezoid is given by $\dfrac{h[f(x_{i-1}) + f(x_i)]}{2}$. If we do this process for each subdivision and sum the area of all the trapezoids, we obtain an approximation of the value of the definite integral (in this case π).
In pursuit of a precise approximation of the area under the curve (when coding), we pick a large number of subdivisions ($n = 1000$) which will in turn provide a greater approximation of π. The following is a modified version of Peter Pacheco's `MPI-C` code for approximating an integral using the trapezoidal method [1].

```
/* File: mpi_trap_time.c
 *
 * Purpose: Implement parallel trapezoidal rule and determine its
 * run-time vs. serial trap rule
 *
 * Input: a, b, n
 * Output: Estimate of the area from between x = a, x = b, x-axis, and
 * the graph of f(x) using the trapezoidal rule and n trapezoids.
```

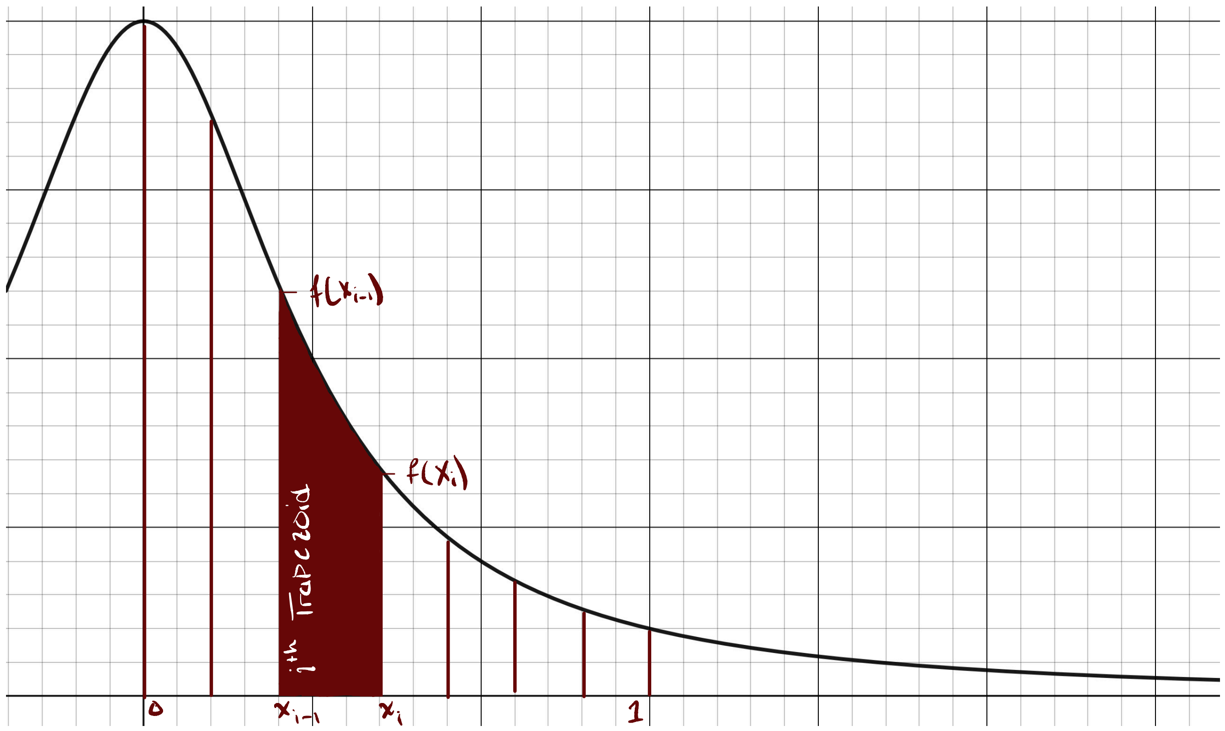


FIGURE 1. Graphical Representation of i^{th} trapezoid

```
 * Also output the elapsed time to run the parallel version.
 *
 * Compile: mpicc -g -Wall -o mpi_trap_time mpi_trap_time.c -lm
 * Run: mpiexec -n <number of processes> ./mpi_trap_time
 *
 * Algorithm:
 * 0. Process 0 reads in a, b, and n, and distributes them
 * among the processes.
 * 1. Barrier.
 * 2. Start timer on each process.
 * 3. Each process calculates "its" subinterval of
 * integration.
 * 4. Each process estimates the area of f(x)
 * over its interval using the trapezoidal rule.
 * 5a. Each process != 0 sends its area to 0.
 * 5b. Process 0 sums the calculations received from
 * the individual processes and prints the result.
 * 6. Stop timer on each process.
 * 7. Find max time, store on process 0.
 * 8. Time serial trap on process 0.
 * 9. Print speedup, efficiency.
 *
 * Note: f(x) is hardwired.
 */
#include <stdio.h>
#include <math.h>

/* We'll be using MPI routines, definitions, etc. */
#include <mpi.h>

void Get_data(int p, int my_rank, double* a_p, double* b_p, int* n_p);

double Trap(double local_a, double local_b, int local_n,
          double h); /* Calculate local area */

double f(double x); /* function we're integrating */

double Get_max_time(double par_elapsed, int my_rank, int p);

int main(int argc, char** argv) {
    int my_rank; /* My process rank */
    int p; /* The number of processes */
    double a; /* Left endpoint */
    double b; /* Right endpoint */
    int n; /* Number of trapezoids */
    double h; /* Trapezoid base length */
    double local_a; /* Left endpoint my process */
    double local_b; /* Right endpoint my process */
    int local_n; /* Number of trapezoids for */
                           /* my calculation */
    double area; /* My subarea */
    double total = 0; /* Total area */
    int source; /* Process sending area */
    int dest = 0; /* All messages go to 0 */
    int tag = 0;
    MPI_Status status;
    double start, finish, par_elapsed;

    /* Let the system do what it needs to start up MPI */
    MPI_Init(&argc, &argv);

    /* Get my process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out how many processes are being used */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    Get_data(p, my_rank, &a, &b, &n);

    MPI_Barrier(MPI_COMM_WORLD);
    start = MPI_Wtime();
    h = (b-a)/n; /* h is the same for all processes */
    local_n = n/p; /* So is the number of trapezoids */

    /* Length of each process' interval of
     * integration = local_n*h. So my interval
     * starts at: */
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    area = Trap(local_a, local_b, local_n, h);

    /* Add up the areas calculated by each process */
    if (my_rank == 0) {
        total = area;
        for (source = 1; source < p; source++) {
            MPI_Recv(&area, 1, MPI_DOUBLE, source, tag,
                MPI_COMM_WORLD, &status);
            total = total + area;
```

```c
        }
    } else {
        MPI_Send(&area, 1, MPI_DOUBLE, dest,
            tag, MPI_COMM_WORLD);
    }
    finish = MPI_Wtime();

    par_elapsed = finish - start;

    par_elapsed = Get_max_time(par_elapsed, my_rank, p);

    /* Print the result */
    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n",
            n);
       // printf("of the area from %f to %f = %1.200f\n",
         // a, b, 4*total);
printf("Pi = %.*f\n",100, 4*total);
        printf("Parallel elapsed time = %e seconds\n", par_elapsed);
    }

    /* Shut down MPI */
    MPI_Finalize();

    return 0;
} /* main */

/*--------------------------------------------------------------------
 * Function: Get_data
 * Purpose: Read in the data on process 0 and send to other
 * processes
 * Input args: p, my_rank
 * Output args: a_p, b_p, n_p
 */
void Get_data(int p, int my_rank, double* a_p, double* b_p, int* n_p) {
   int q;
   MPI_Status status;

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);

      for (q = 1; q < p; q++) {
         MPI_Send(a_p, 1, MPI_DOUBLE, q, 0, MPI_COMM_WORLD);
         MPI_Send(b_p, 1, MPI_DOUBLE, q, 0, MPI_COMM_WORLD);
         MPI_Send(n_p, 1, MPI_INT, q, 0, MPI_COMM_WORLD);
      }
   } else {
      MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
      MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
      MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
   }
} /* Get_data */

/*--------------------------------------------------------------------
```

```
 * Function: Trap
 * Purpose: Estimate a definite area using the trapezoidal
 * rule
 * Input args: local_a (my left endpoint)
 * local_b (my right endpoint)
 * local_n (my number of trapezoids)
 * h (stepsize = length of base of trapezoids)
 * Return val: Trapezoidal rule estimate of area from
 * local_a to local_b
 */
double Trap(
         double local_a /* in */,
         double local_b /* in */,
         int local_n /* in */,
         double h /* in */) {
   double area; /* Store result in area */
   double x;
   int i;

   area = (f(local_a) + f(local_b))/2.0;
   x = local_a;
   for (i = 1; i <= local_n-1; i++) {
       x = local_a + i*h;
       area = area + f(x);
   }
   area = area*h;

   return area;
} /* Trap */

/*------------------------------------------------------------------
 * Function: f
 * Purpose: Compute value of function to be integrated
 * Input args: x
 */
double f(double x) {
   double return_val;

// return_val = x*x;
// return_val = atan(x);
    return_val = 1/(1+pow(x,2));
   return return_val;
} /* f */


/*------------------------------------------------------------------
 * Function: Get_max_time
 * Purpose: Find the maximum elapsed time across the processes
 * In args: my_rank: calling process' rank
 * p: total number of processes
 * par_elapsed: elapsed time on calling process
 * Ret val: Process 0: max of all processes times
 * Other procs: input value for par_elapsed
 */
double Get_max_time(double par_elapsed, int my_rank, int p) {
   int source;
   MPI_Status status;
   double temp;

   if (my_rank == 0) {
      for (source = 1; source < p; source++) {
         MPI_Recv(&temp, 1, MPI_DOUBLE, source, 0, MPI_COMM_WORLD, &status);
         if (temp > par_elapsed) par_elapsed = temp;
      }
   } else {
      MPI_Send(&par_elapsed, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
   }
   return par_elapsed;
} /* Get_max_time */
```

Figure 2 shows the number of processes versus the wall time needed to complete the calculation of π. We can clearly see a performance improvement as we increase the number of processes p from 1 to 3. However, as the number of processes increases past 3 performance starts to degrade; this is likely due to the overhead communication time that `MPI` needs. It is interesting to note that performance starts to improve at $p = 8$, which may or may not be an anomaly.

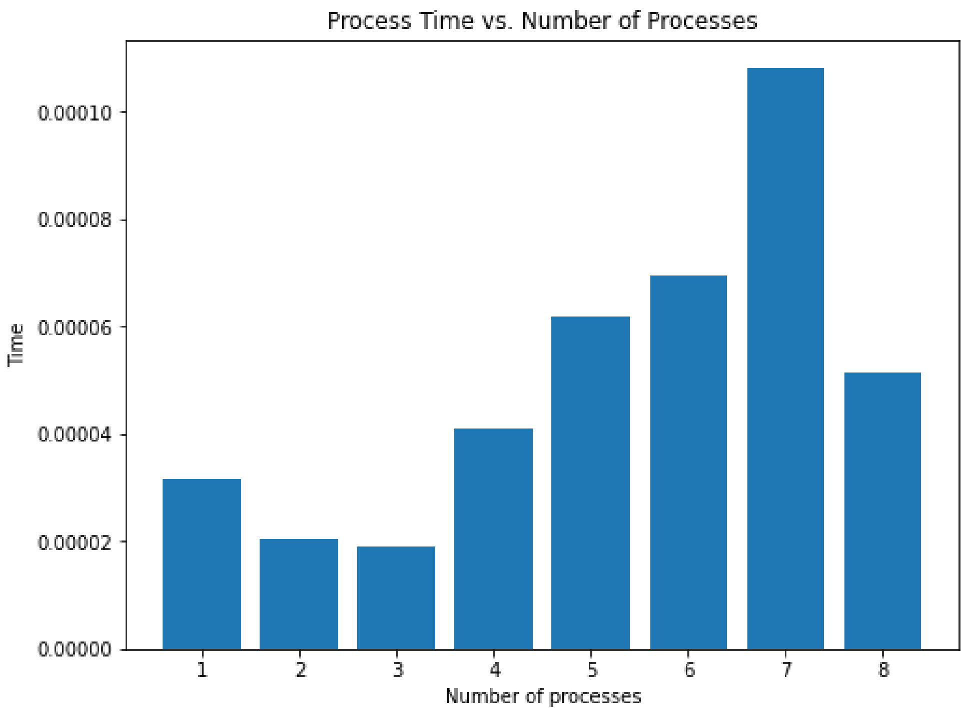


FIGURE 2. Computational run time versus the number of processes for approximating π using the trapezoidal rule.

TASK 4

The Lotka Volterra equations are a system of first order nonlinear differential equations that describe the dynamics between a predator y and prey x in an ecological system. For $\alpha, \beta \in \mathbb{R}^+$, the system is given by:

$$\dot{x} = x(\alpha - \beta y)$$
$$\dot{y} = y(\gamma x - \delta)$$

By collapsing the system above to a single equation and using separation of variables, we integrate:

$$\int \frac{\beta y - a}{y} dy + \int \frac{\delta x - \gamma}{x} dx = 0$$

will yield some constant $C = \delta x - \gamma \ln(x) + \beta y - a \ln(y)$. This constant will be recovered using the initial values provided for both the predator and prey at time $t = 0$.
The following is a minimal example of parallelizing the Lotka Volterra equations in `Python` using `Pathos`, a library aimed at easing the overhead of implementing parallel computations. Using ProcessPool to indicate the number of processes to be used for each execution, and Scipy's `odeint` function. Please note that this code was made in tandem with `chatGPT` in order to use distributed computing.

```python
import time
import numpy as np
import numba as nb
import matplotlib.pyplot as plt
from scipy.integrate import odeint
from pathos.multiprocessing import ProcessPool as Pool

# Define the function for parallel solving
def solve_parallel(processes):
    # Define the Lotka-Volterra function with numba.jit
    @nb.njit
    def lotka_volterra(u, t):
        # Define the parameters for the differential equations
        prey_growth = 4.0
        predation_rate = 3.0
        predator_growth = 5.0
        predator_decay = 7.0
        prey, predator = u

        prey_dt = prey_growth * prey - predation_rate * prey * predator
        predator_dt = predator_growth * prey * predator - predator_decay * predator

        return np.array([prey_dt, predator_dt])

    # Define the initial conditions
    initial_prey = 4.0
    initial_predator = 2.0
    t_eval = np.linspace(0.0, 30.0, 3001)
    n = 10000
    prey_init = np.random.uniform(initial_prey, initial_prey, n).reshape((n, 1))
    predator_init = np.random.uniform(initial_predator, initial_predator, n).reshape
        ((n, 1))
    initial_conditions_all = np.append(prey_init, predator_init, axis=1)

    # Define the solver function that takes initial conditions and integrates using
       scipy odeint
    def solver(initial_conditions):
        solution = odeint(lotka_volterra, initial_conditions, t_eval)
        return solution[:, 0], solution[:, 1]




    # Create a process pool and run the solver in parallel
    start_time = time.time()

    p = Pool(processes)
    solutions = p.map(solver, initial_conditions_all)
    end_time = time.time()
    elapsed_time = end_time - start_time

    # Separate the prey and predator populations from the solutions
    prey_population = np.empty((n, len(t_eval)), np.float64)
    predator_population = np.empty((n, len(t_eval)), np.float64)
    for i in range(n):
        prey_population[i] = solutions[i][0]
        predator_population[i] = solutions[i][1]

    # Plot the results
    plt.figure()
    plt.grid()
    plt.plot(t_eval, np.mean(prey_population, axis=0), 'xb', label='Prey')
    plt.plot(t_eval, np.mean(predator_population, axis=0), '+r', label='Predator')
    plt.xlabel('Days')
    plt.ylabel('Population')
    plt.legend()
    plt.show()
    plt.figure()
    initial_conditions = [initial_prey, initial_predator]
    populations = odeint(lotka_volterra, initial_conditions, t_eval)
    plt.plot(populations[:, 0], populations[:, 1], "-")
    plt.xlabel("Prey")
    plt.ylabel("Predator")
    plt.title("Prey vs Predator Phase Portrait")
    plt.show()

    print("Elapsed time for initial predator population = " + str(initial_predator) +
         ": " + str(elapsed_time) + " seconds")

# Run the loop for processes in range(1, 9)
for processes in range(1, 9):
    solve_parallel(processes)
```

Figures 3 & 4 show a scatter plot of the population as time increases and a phase portrait of the dynamics of the ecological system. Figure 5 shows the number of processes versus the wall time needed to complete an approximation of solution to the Lotka Volterra system of differential equations.

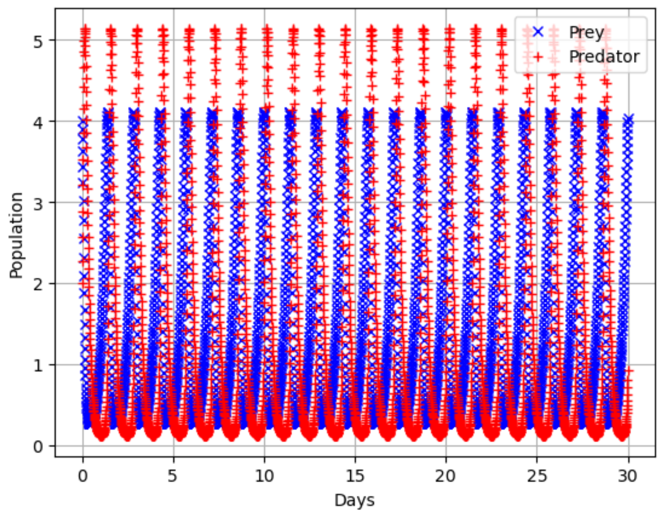


FIGURE 3. Scatter plot.

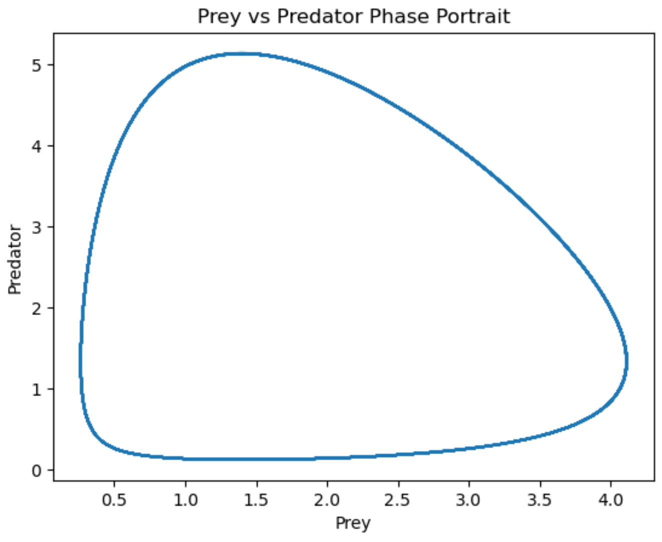


FIGURE 4. Phase portait.

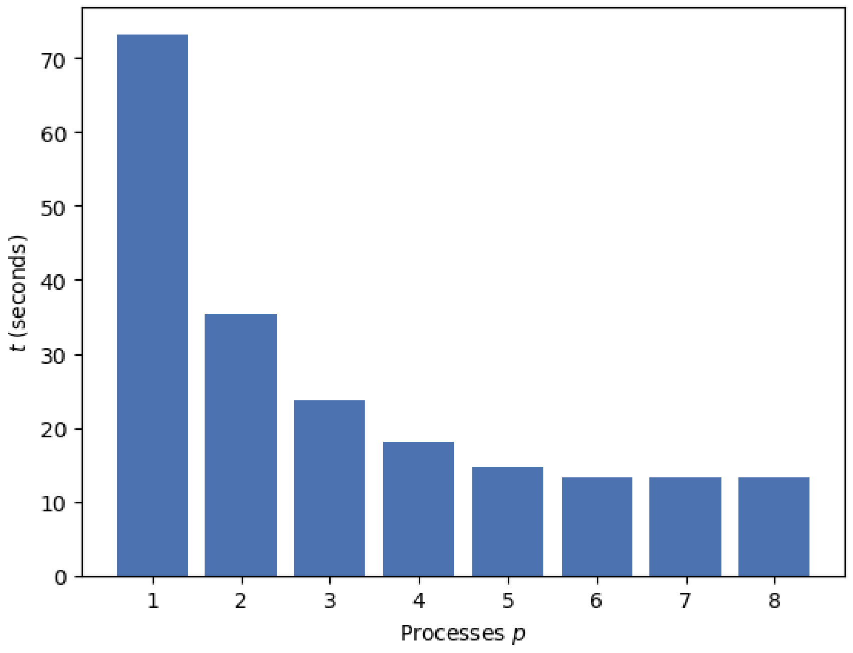


FIGURE 5. Computational run times versus the number of processes p for the Lotka Volterra system of differential equations.

References

[1] Peter S. Pacheco. *Parallel programming with MPI*. Kaufmann, 2000.