**Overview**

In this lab, you’ll explore **computer parallelism** by running experiments on a Raspberry Pi. You’ll work with multiple levels of parallelism—from bit-level optimizations to distributed computing using a Raspberry Pi cluster.

By the end of this lab, you will be able to:

* Identify and differentiate types of computer parallelism.
* Implement task-level parallelism using Python’s multiprocessing module.
* Use vectorized operations (SIMD) for data-level parallelism.
* Profile instruction-level parallelism with performance tools.
* Configure a multi-node Raspberry Pi cluster and run distributed programs using MPI.

**Requirements**

* One or more Raspberry Pi 3, 4, or 5 with Raspbian OS (64-bit preferred)
* Python 3 and pip (Thorny)
* C compiler (gcc) (Code Blocks)
* Libraries:  
  sudo apt update

sudo apt install mpich python3-mpi4py linux-perf build-essential

pip3 install numpy

**Lab Exercises**

**Task 1: Bit-Level Parallelism (Data Width Impact)**

**🔧 Setup**

1. In Code Blocks create a C file:  
    bit\_parallel.c

1. Add the following code to compare 8-bit, 16-bit, and 32-bit operations:

|  |
| --- |
| #include <stdio.h>  #include <stdint.h>  #include <time.h>  void test\_type(const char\* label, int iterations) {  uint64\_t sum = 0;  clock\_t start = clock();  for (int i = 0; i < iterations; i++) {  **uint32\_t a = i;**  sum += a \* a;  }  clock\_t end = clock();  printf("%s: Time = %fs\n", label, (double)(end - start)/CLOCKS\_PER\_SEC);  }  int main() {  test\_type("32-bit Integer", 100000000);  return 0;  } |

1. Compile and run:

**📝 Observe**

Compare the performance using different data widths (e.g., 8-bit, 16-bit, 32-bit). Try modifying a to use different types (uint8\_t, uint16\_t, etc.). You will substitute these values in the highlighted line.  
Record your results.

**Task 2: Instruction-Level Parallelism (ILP Profiling)**

**🔧 Setup**

1. Write a simple loop in C:

File name: ilp\_test.c

|  |
| --- |
| #include <stdio.h>  int main() {  volatile int sum = 0;  for (int i = 0; i < 100000000; i++) {  sum += i;  }  return 0;  } |

1. Compile and run with performance profiling:   
    Do this from the cmd line.  
     
   gcc -O2 ilp\_test.c -o ilp\_test

perf stat ./ilp\_test

**📝 Observe**

Note the **instructions per cycle (IPC)**, branch misses, and cache hits. Discuss how the CPU’s ILP mechanisms handle loop execution.

**Task 3: Task-Level Parallelism (Python Multiprocessing)**

**🔧 Setup**

1. Create a Python file in Thorny

parallel\_fibonacci.py

1. Enter the following code:

|  |
| --- |
| Import time  Import multiprocessing  def fibonacci(n):  if n <= 1:  return n  return fibonacci(n - 1) + fibonacci(n - 2)  def parallel\_fibonacci(n):  result = fibonacci(n)  print(f"Fibonacci({n}) = {result}")  return result  if \_\_name\_\_ == "\_\_main\_\_":  fib\_numbers = [30, 31, 32, 33]  print("Sequential...")  start = time.time()  [fibonacci(n) for n in fib\_numbers]  print(f"Time: {time.time() - start:.2f} seconds\n")  print("Parallel...")  start = time.time()  with multiprocessing.Pool(processes=multiprocessing.cpu\_count()) as pool:  pool.map(parallel\_fibonacci, fib\_numbers)  print(f"Time: {time.time() - start:.2f} seconds\n") |

1. Run it:

**Observe**

Compare execution time between sequential and parallel computation. Calculate the **speedup factor**. Speedup (SF) = Time Sequential / Time Parallel

**Task 4: Data-Level Parallelism (NumPy Matrix Multiplication)**

**Setup:**

1. Create a Python file:  
   matric\_mult.py

|  |
| --- |
| import numpy as np  import time  size = 500  A = np.random.rand(size, size)  B = np.random.rand(size, size)  start = time.time()  result = A @ B  print(f"Vectorized Time: {time.time() - start:.2f} seconds")  result = [[sum(a \* b for a, b in zip(A\_row, B\_col)) for B\_col in B.T] for A\_row in A]  print(f"Scalar Time: {time.time() - start:.2f} seconds") |

1. Run it:

**Observe**

Note the significant time difference between vectorized and scalar versions, showing the power of **SIMD and NumPy**.

**🔹 Task 5: Distributed Computing with MPI (Raspberry Pi Cluster)**

**Setup**

**Note: This will be done in class on the day of the lab**

1. Connect Raspberry Pis to the same router/switch via ethernet
2. Assign static IP address

On each RPi you will edit the dhcpcd.conf file

From the cmd line: sudo nano /etc/dhcpcd.conf

Add the following at the end:

|  |
| --- |
| interface eth0  **static ip\_address=192.168.1.100/24**  static routers=192.168.1.1  static domain\_name\_servers=192.168.1.1 |

Change the highlighted line for an address specific to each RPi

|  |
| --- |
| Pi 1: 192.168.1.101/24  Pi 2: 192.168.1.102/24  Pi 3: 192.168.1.103/24  Pi 4: 192.168.1.104/24  Pi 5: 192.168.1.104/24  Pi 6: 192.168.1.105/24  etc |

The subnet mask /24 is shorthand for 255.255.255.0.

1. You will also have to give your RPi a hostname:

We will use rpi1 – rpi6 you will be assigned a hostname in class.

sudo hostnamectl set-hostname rpi0

1. Reboot: sudo reboot
2. Test communications:

Connect each RPi to the switch via the ethernet cable  
You should be able to ping to any of the other RPis

From the cmd line: pint 192.168.1.100

Do this for each of the other RPIs 101 – 106

1. Install MPI:

sudo apt install mpich python3-mpi4py

1. Create: mpi\_primes.py:
2. Enter this code:

|  |
| --- |
| from mpi4py import MPI  import math  def is\_prime(n):  if n <= 1:  return False  for i in range(2, int(math.sqrt(n)) + 1):  if n % i == 0:  return False  return True  comm = MPI.COMM\_WORLD  rank = comm.Get\_rank()  size = comm.Get\_size()  start = rank \* 10000 // size  end = (rank + 1) \* 10000 // size  local\_primes = [n for n in range(start, end) if is\_prime(n)]  all\_primes = comm.gather(local\_primes, root=0)  if rank == 0:  primes = [p for sublist in all\_primes for p in sublist]  print(f"Found {len(primes)} primes.") |

1. This will run on 2 nodes:

mpirun -np 2 -host rpi1,rpi2 python3 mpi\_primes.py

**Observe**

Distributed execution across multiple Pis demonstrates **scalability** and the concept of **distributed memory**. Run this using different node configurations

**After completing the lab and recording your results answer these Questions**

1. For each of the following examples, identify the type of parallelism being used and explain your reasoning:
   * A Python script using the multiprocessing module to run 4 independent tasks.
   * Matrix multiplication accelerated with NumPy’s vectorized operations.
   * A Raspberry Pi cluster using MPI to calculate parts of a large dataset.
2. If your calculation rand sequentially in 18.0 ms and in parallel on 4 cores in 5.2 seconds
   * What is the speedup factor?
   * Is this an ideal speedup factor
   * What factors could prevent a perfect 4x improvement?
3. Which type of parallelism showed the greatest performance gain?
4. How does processor architecture (e.g., SIMD vs multicore) affect performance?
5. What are some real-world tasks that would benefit from each type of parallelism?
6. You’ve configured various numbers of RPis to compute prime numbers in parallel using MPI. Each node is responsible for an equal portion of the range 1–1,000,000.

After running the program, you notice that **rpi0 completes its task much slower** than the other nodes, despite all Pis having the same hardware and workload.

What are three possible reasons for this performance imbalance, and how would you diagnose and resolve the issue?