## Indian Institute of Information Technology Surat

****

# Lab Report on

# High Performance Computing (CS 602) Practical

**Submitted by**

### [RAHUL KUMAR SINGH] (UI21CS44)

**Course Faculty**

### Dr. Sachin D. Patil

## Department of Computer Science and Engineering

## Indian Institute of Information Technology Surat

## Gujarat-394190, India

**Jan-2024**

## Lab No: 1

**Aim: Write a parallel program in c, c++, java, or python to compute the dot product and cross product for N elements vector arrays a[N] and b[N].**

**Description:** Steps to follow:

* Divide the vectors into smaller chunks, assigning each chunk to a separate thread.
* Each thread independently computes the cross product for its assigned chunk.
* Leverage parallel processing to enhance overall computation speed.
* The cross product involves multiplying corresponding elements (when n < 4 apply the directional cross product).
* Sum all the products to obtain the dot product.

## Source Code:

import threading

import multiprocessing

import random

def generate\_random\_vector(n):

return [random.randint(1, 10) for \_ in range(n)]

def compute\_partial\_dot\_product(start, end, result, a, b):

partial\_sum = 0

for i in range(start, end):

partial\_sum += a[i] \* b[i]

result.append(partial\_sum)

def parallel\_dot\_product(a, b, num\_threads):

n = len(a)

step = n // num\_threads

result = []

threads = []

for i in range(num\_threads):

start = i \* step

end = (i + 1) \* step if i < num\_threads - 1 else n

thread = threading.Thread(target=compute\_partial\_dot\_product, args=(start, end, result, a, b))

threads.append(thread)

thread.start()

for thread in threads:

thread.join()

dot\_product = sum(result)

return dot\_product

def compute\_cross\_product(start, end, a, b, result):

for i in range(start, end):

result[i] = a[i] \* b[i]

def parallel\_cross\_product(a, b):

n = len(a)

result = multiprocessing.Array('d', n)

num\_processes = multiprocessing.cpu\_count()

chunk\_size = n // num\_processes

processes = []

for i in range(num\_processes):

start = i \* chunk\_size

end = (i + 1) \* chunk\_size if i != num\_processes - 1 else n

process = multiprocessing.Process(target=compute\_cross\_product, args=(start, end, a, b, result))

processes.append(process)

for process in processes:

process.start()

for process in processes:

process.join()

return list(result)

if \_\_name\_\_ == "\_\_main\_\_":

N = 10

a = generate\_random\_vector(N)

b = generate\_random\_vector(N)

print(f"Vector a: {a}")

print(f"Vector b: {b}")

num\_threads = 2

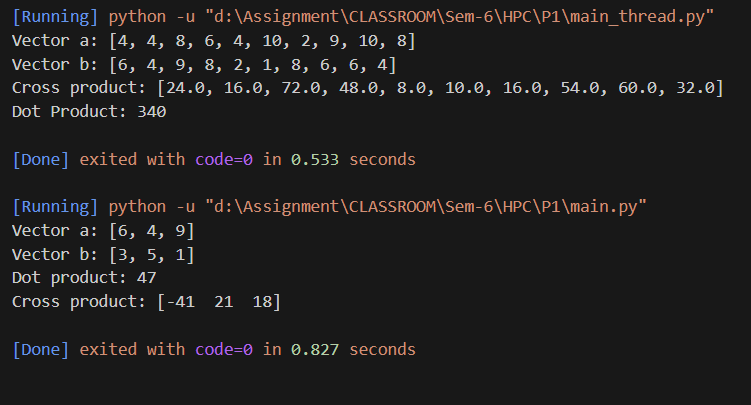
result = parallel\_cross\_product(a, b)

print(f"Cross product: {result}")

result = parallel\_dot\_product(a, b, num\_threads)

print(f"Dot Product: {result}")

## Output:

****

## Conclusion:

* Leveraging threads for both cross product and dot product computation enables parallel processing, optimizing performance for large N.
* Incorporating random inputs adds realism to the computational model, simulating scenarios with varied data distributions.