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()
```

```python
    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}")
```

```python
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:

```
[Running] python -u "d:\Assignment\CLASSROOM\Sem-6\HPC\P1\main_thread.py"
Vector a: [4, 4, 8, 6, 4, 10, 2, 9, 10, 8]
Vector b: [6, 4, 9, 8, 2, 1, 8, 6, 6, 4]
Cross product: [24.0, 16.0, 72.0, 48.0, 8.0, 10.0, 16.0, 54.0, 60.0, 32.0]
Dot Product: 340

[Done] exited with code=0 in 0.533 seconds

[Running] python -u "d:\Assignment\CLASSROOM\Sem-6\HPC\P1\main.py"
Vector a: [6, 4, 9]
Vector b: [3, 5, 1]
Dot product: 47
Cross product: [-41  21  18]

[Done] exited with code=0 in 0.827 seconds
```

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.