Author: Pupipat Singkhorn, 6532142421

Question 1

What is the main benefit of openmp over standard thread (eg. POSIX thread)?

Answer

OpenMP provides a higher-level abstraction for parallel programming compared to POSIX threads (Pthreads). The key benefits of OpenMP over Pthreads include:

- **Ease of Use**: OpenMP uses compiler directives (#pragma omp . . .) that make parallelization simpler without manually managing threads.
- **Automatic Work Distribution**: OpenMP automatically distributes work among available threads, whereas Pthreads require explicit thread management.
- **Better Scalability**: OpenMP can dynamically adjust thread counts based on system resources.
- **Portability**: OpenMP code is easily portable across different architectures and compilers that support OpenMP.

Question 2

For a given code, modify it to take the benefit of simultaneous multithreading processor using openmp. With the new code, what is the potential speed up?

```c
#include <stdio.h>
int main(void)
{
int a[100000];
for (int i=0;i<100000;i++) {
a[i]=2*i+i;
printf("a[%d],%d\n",i, a[i]);
}
return 0;
}
```

Answer

Experiment setup:

- Iterations: 3
- Array Sizes
  - SIZE1 = 100000 / 100
  - SIZE2 = 100000 (default)

- SIZE3 = 100000 * 100

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define ITERATIONS 3  // Number of times to run each test
#define SIZE1 100000 / 100
#define SIZE2 100000 // Default size
#define SIZE3 100000 * 100

void serial_version(int size, double *serial_time) {
    int *a = (int*)malloc(size * sizeof(int)); // Allocate memory for large arrays
    if (!a) {
        printf("Memory allocation failed for size %d\n", size);
        exit(1);
    }

    double start_time = omp_get_wtime();
    for (int i = 0; i < size; i++) {
        a[i] = 2 * i + i; // Computes 3*i
    }
    double end_time = omp_get_wtime();

    *serial_time = (end_time - start_time) * 1000.0; // Convert to milliseconds
    free(a); // Free memory
}

void parallel_version(int size, double *parallel_time) {
    int *a = (int*)malloc(size * sizeof(int)); // Allocate memory for large arrays
    if (!a) {
        printf("Memory allocation failed for size %d\n", size);
        exit(1);
    }

    double start_time = omp_get_wtime();
    #pragma omp parallel for
    for (int i = 0; i < size; i++) {
        a[i] = 2 * i + i;
    }
    double end_time = omp_get_wtime();

    *parallel_time = (end_time - start_time) * 1000.0; // Convert to milliseconds
    free(a); // Free memory
}
```

```c
void run_test(int size) {
    printf("\n=== Testing with SIZE = %d ===\n", size);

    double total_serial = 0.0, total_parallel = 0.0;

    for (int i = 0; i < ITERATIONS; i++) {
        double serial_time, parallel_time;

        serial_version(size, &serial_time);
        parallel_version(size, &parallel_time);

        total_serial += serial_time;
        total_parallel += parallel_time;

        printf("  Iteration %d: Serial = %.3f ms, Parallel = %.3f ms, Speedup = %.2fX\n",
               i + 1, serial_time, parallel_time, serial_time / parallel_time);
    }

    double avg_serial = total_serial / ITERATIONS;
    double avg_parallel = total_parallel / ITERATIONS;
    double avg_speedup = avg_serial / avg_parallel;

    printf("\n  >>> AVERAGE RESULTS for SIZE = %d <<<\n", size);
    printf("  Avg Serial Time: %.3f ms\n", avg_serial);
    printf("  Avg Parallel Time: %.3f ms\n", avg_parallel);
    printf("  Avg Speedup: %.2fX\n", avg_speedup);
}

int main(void) {
    int num_cores = omp_get_num_procs();
    printf("Number of CPU cores: %d\n", num_cores);

    run_test(SIZE1);
    run_test(SIZE2);
    run_test(SIZE3);

    return 0;
}
```

```bash
gcc-14 -fopenmp -o q2 q2.c
./q2
```

Results:

```
Number of CPU cores: 8

=== Testing with SIZE = 1000 ===
  Iteration 1: Serial = 0.005 ms, Parallel = 0.194 ms, Speedup = 0.03X
  Iteration 2: Serial = 0.005 ms, Parallel = 0.067 ms, Speedup = 0.07X
  Iteration 3: Serial = 0.005 ms, Parallel = 0.071 ms, Speedup = 0.07X

  >>> AVERAGE RESULTS for SIZE = 1000 <<<
  Avg Serial Time: 0.005 ms
  Avg Parallel Time: 0.111 ms
  Avg Speedup: 0.05X

=== Testing with SIZE = 100000 ===
  Iteration 1: Serial = 0.508 ms, Parallel = 0.319 ms, Speedup = 1.59X
  Iteration 2: Serial = 0.502 ms, Parallel = 0.230 ms, Speedup = 2.18X
  Iteration 3: Serial = 0.502 ms, Parallel = 0.229 ms, Speedup = 2.19X

  >>> AVERAGE RESULTS for SIZE = 100000 <<<
  Avg Serial Time: 0.504 ms
  Avg Parallel Time: 0.259 ms
  Avg Speedup: 1.94X

=== Testing with SIZE = 10000000 ===
  Iteration 1: Serial = 45.918 ms, Parallel = 4.953 ms, Speedup = 9.27X
  Iteration 2: Serial = 27.811 ms, Parallel = 5.230 ms, Speedup = 5.32X
  Iteration 3: Serial = 23.906 ms, Parallel = 5.426 ms, Speedup = 4.41X

  >>> AVERAGE RESULTS for SIZE = 10000000 <<<
  Avg Serial Time: 32.545 ms
  Avg Parallel Time: 5.203 ms
  Avg Speedup: 6.26X
```

Conclusion

The experiment demonstrates that OpenMP parallelization significantly improves performance for **large workloads** but introduces **overhead** for small tasks.

Question 3

Base on OpenMP, explain the concepts of work sharing constructs for

- loop constructs: for and do
- sections
- single
- Workshare

Answer

- **Loop Constructs (`for`, `do`):** Distributes loop iterations across threads. Example:

```
#pragma omp parallel for
for (int i = 0; i < N; i++) { ... }
```

- **Sections:** Divides code into independent blocks for parallel execution. Example:

```
#pragma omp parallel sections
{
  #pragma omp section
  { ... } // Task 1
  #pragma omp section
  { ... } // Task 2
}
```

- **Single:** Ensures a code block is executed by only one thread. Example:

```
#pragma omp single
{ printf("This runs once\n"); }
```

- **Workshare (Fortran-specific):** Parallelizes array operations and `FORALL/WHERE` statements in Fortran. Not applicable to C/C++. Example:

```
!$OMP WORKSHARE
A = B + C
!$OMP END WORKSHARE
```

References

- CURC, Using OpenMP with C
- OpenMP, OpenMP Application Programming Interface
- ChatGPT
- DeepSeek