Day5

Table of Contents

1. OpenMP Directives

- `#pragma omp parallel`: Defines a parallel region.
- `#pragma omp for`: Distributes loop iterations among threads.

Example:

```c
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    a[i] = b[i] + c[i];
}
```

2. OpenMP Clauses

- private(var): Each thread has its own copy of the variable.
- shared(var): The variable is shared among all threads.
- reduction(op:var): Combines values from all threads using the specified operation.

Example:

```c
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < n; i++) {
    sum += a[i];
}
```

3. Environment Variables

- OMP_NUM_THREADS: Sets the number of threads.
- OMP_DYNAMIC: Enables or disables dynamic adjustment of threads.
- OMP_SCHEDULE: Controls the schedule type for loops (e.g., static, dynamic).

Example:

```bash
export OMP_NUM_THREADS=4
export OMP_SCHEDULE="dynamic"
```

4. OpenMP Directives and Clauses

4.1. OpenMP Directives

OpenMP directives are instructions added to the code to enable parallel execution. They are identified by the `#pragma omp` keyword and guide the compiler to parallelize sections of the program.

```
#pragma omp directive [clauses]
```

4.2. Common OpenMP Directives

- #pragma omp parallel Defines a parallel region where multiple threads execute the code block.
- #pragma omp for Distributes iterations of a loop among threads for parallel execution.
- #pragma omp sections Divides the program into sections where different threads execute different blocks.
- #pragma omp single Ensures a block of code is executed by only one thread.
- #pragma omp critical Protects a block of code so that only one thread executes it at a time.

4.3. Example of OpenMP Directives

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

int main() {
    #pragma omp parallel
    {
        printf("Hello from thread %d\n", omp_get_thread_num());
    }
    return 0;
}
```

In this example, the `#pragma omp parallel` directive creates a parallel region, and each thread prints its thread ID.

_

5. Clauses in OpenMP

Clauses modify the behavior of OpenMP directives. They are used to control data sharing, schedule loops, and synchronize operations.

5.1. Common OpenMP Clauses

- private(variable): Each thread has its own private copy of the variable.
- shared(variable): The variable is shared among all threads.
- reduction(operator:variable): Performs a reduction operation (e.g., sum, product) across all threads.
- firstprivate(variable): Each thread gets a private copy of the variable, initialized with the value from the master thread.
- schedule(type[, chunk]): Specifies how loop iterations are divided among threads.

6. Data Scopes in OpenMP

OpenMP allows control over the visibility and scope of variables in a parallel region.

6.1. Types of Data Scopes

- **shared**: The variable is shared among all threads.
- **private**: Each thread has its own private copy of the variable.
- **firstprivate**: Each thread gets a private copy of the variable, initialized with the master thread's value.
- **lastprivate**: Updates the value of a private variable back to the shared variable after the loop ends.

6.2. Example: Data Scopes

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

```c
int main() {
    int x = 10;

    #pragma omp parallel private(x)
    {
        x = omp_get_thread_num();
        printf("Thread %d, x = %d\n", omp_get_thread_num(), x);
    }
    printf("Outside parallel region, x = %d\n", x);
    return 0;
}
```

—

7. OpenMP Constructs

OpenMP constructs are building blocks for writing parallel code.

7.1. Common Constructs

- Parallel Region:
  - `#pragma omp parallel`: Creates a team of threads.
- Work Sharing Constructs:
  - `#pragma omp for`: Distributes loop iterations.
  - `#pragma omp sections`: Divides tasks into separate code blocks.
  - `#pragma omp single`: Ensures a block is executed by one thread.
  - `#pragma omp master`: Only the master thread executes the block.
  - `#pragma omp critical`: Protects critical sections of code.

7.2. Example: Work Sharing Constructs

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

int main() {
    #pragma omp parallel
    {
```

```
        #pragma omp single
        printf("Only one thread executes this.\n");

        #pragma omp for
        for (int i = 0; i < 8; i++) {
            printf("Thread %d, iteration %d\n", omp_get_thread_num(), i);
        }
    }
    return 0;
}
```

8. private

```
#include<stdio.h>
#include<omp.h>
int main(){
    int a = 5;
    #pragma omp parallel private(a) num_threads(4)
    {
        printf("Inside: a = %d by tid %d\n", a, omp_get_thread_num());
    }
    printf("After: a = %d\n", a);
    return 0;
}
```

```
gcc test.c -fopenmp
```

```
./a.out
```

```
Inside: a = 5 by tid 1 : 0x7ffe7faa33f8
Inside: a = 5 by tid 2 : 0x7ffe7faa33f8
Inside: a = 5 by tid 0 : 0x7ffe7faa33f8
Inside: a = 5 by tid 3 : 0x7ffe7faa33f8
After: a = 5 : 0x7ffe7faa33f8
```

9. firstprivate

```c
#include<stdio.h>
#include<omp.h>
int main(){
    int a = 5;
    #pragma omp parallel firstprivate(a) num_threads(4)
    {
        printf("Inside: a = %d by tid %d : %p\n", a, omp_get_thread_num(), &a);
    }
    printf("After: a = %d : %p\n", a, &a);
    return 0;
}
```

```
gcc firstprivate.c -fopenmp
```

```
./a.out
```

```
Inside: a = 5 by tid 3 : 0x7ab901f8ede4
Inside: a = 5 by tid 2 : 0x7ab90298ede4
Inside: a = 5 by tid 0 : 0x7ffc941ee4e4
Inside: a = 5 by tid 1 : 0x7ab90338ede4
After: a = 5 : 0x7ffc941ee530
```

10. default

```c
#include<stdio.h>
#include<omp.h>
int main(){
    int a = 5;
    int b = 234;
    #pragma omp parallel default(none) shared(a) private(b) num_threads(4)
    {
        printf("Inside: a = %d by tid %d : %p\n", a, omp_get_thread_num(), &a);
        b = 234;
    }
    printf("After: a = %d : %p\n", a, &a);
    return 0;
}
```

```
gcc default.c -fopenmp
```

```
./a.out
```

```
Inside: a = 5 by tid 2 : 0x7ffeb3a11d48
Inside: a = 5 by tid 0 : 0x7ffeb3a11d48
Inside: a = 5 by tid 1 : 0x7ffeb3a11d48
Inside: a = 5 by tid 3 : 0x7ffeb3a11d48
After: a = 5 : 0x7ffeb3a11d48
```

11. Test1

```c
#include<stdio.h>
#include<omp.h>
int main(){
    int a = 5;
    #pragma omp parallel private(a) num_threads(10)
    {
        int tid = omp_get_thread_num();
        if(tid == 3) a = 7;
        printf("Inside: a = %d by tid %d\n", a, tid);
    }
    printf("After: a = %d\n", a);
    return 0;
}
```

```
gcc test1.c -fopenmp
```

```
./a.out
```

```
Inside: a = 23 by tid 9
Inside: a = 23 by tid 6
Inside: a = 23 by tid 1
Inside: a = 7 by tid 3
Inside: a = 23 by tid 7
```

```
Inside: a = 23 by tid 0
Inside: a = 23 by tid 5
Inside: a = 23 by tid 2
Inside: a = 23 by tid 8
Inside: a = 23 by tid 4
After: a = 5
```

12. Task1

Create an array and print the elements of that array inside parallel region. Devide your data between number of threads

```c
#include<stdio.h>
#include<omp.h>
#define N 100000
#define T 10
int main(){
    int a[N];
    for(int i = 0; i < N; i++) a[i] = i + 1;

    #pragma omp parallel num_threads(T)
    {
        for(int i = 0; i < N; i++){
            printf("%d ", a[i]);
        }
        printf("\n");
    }

    return 0;
}
```

```
gcc task1.c -fopenmp -o task1.out
```

```
./task1.out > output1.txt
echo "check output1.txt"
```

```
check output1.txt
```

13. solution task1

Create an array and print the elements of that array inside parallel region.

```
#include<stdio.h>
#include<omp.h>
#define N 100000
#define T 10
int main(){
    int a[N];
    for(int i = 0; i < N; i++) a[i] = i + 1;

    int start, end;
    int chunksize = N / T;
    #pragma omp parallel shared(chunksize) private(start, end) num_threads(T)
    {
        int tid = omp_get_thread_num();
        start = tid * chunksize;
        end = start + chunksize;
        if(tid == T - 1) end = N;
        for(int i = start; i < end; i++){
            printf("%d ", a[i]);
        }
    }

    return 0;
}
```

```
gcc task1_sol.c -fopenmp -o task1_sol.out
```

```
./task1_sol.out > output2.txt
echo "check output2.txt"
```

```
check output2.txt
```

14. Task2

```c
#include<stdio.h>
#include<omp.h>
#define N 21
#define T 10
int main(){
    int a[N];
    for(int i = 0; i < N; i++) a[i] = i + 1;

    #pragma omp parallel num_threads(T)
    {
        #pragma omp for
        for(int i = 0; i < N; i++){
            printf("%d ", a[i]);
        }
    }

    return 0;
}
```

```
gcc task2.c -fopenmp -o task2.out
```

```
./task2.out | wc -w
./task2.out > output3.txt
echo "check output3.txt"
```

```
21
check output3.txt
```

15. Task3

Write a program to calculate sum of natural numbers.

Author: Abhishek Raj
Created: 2024-12-18 Wed 17:12