I3340 - OpenMp

Ali Saab

July 2023

I3340 - OpenMp

Ali Saab

Contents

1 OpenMp

1.1 Parallel Region

```
omp_set_num_threads(nthreads) //Set the number of threads used in a parallel region
omp_get_num_threads() //Get the number of threads used in a parallel region
omp_get_thread_num() //Get the thread rank in a parallel region (0 to omp_get_num_threads() - 1)
```

Example

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

int main() {
    int nthreads, thread_num;

    // Set the number of threads used in a parallel region
    omp_set_num_threads(4);

    #pragma omp parallel
    {
        // Get the number of threads used in the parallel region
        nthreads = omp_get_num_threads();
        // Get the thread rank in the parallel region
        thread_num = omp_get_thread_num();

        printf("Hello from thread %d out of %d threads\n", thread_num, nthreads);
    }

    return 0;
}
Hello from thread 0 out of 4 threads
Hello from thread 1 out of 4 threads
Hello from thread 3 out of 4 threads
Hello from thread 2 out of 4 threads
```

Note:
In OpenMP, when using directives like `#pragma omp parallel for`, the main thread is also one of the threads that participate in the parallel region. The main thread initially acts as the master thread and executes the code before the parallel region. However, once the parallel region is encountered, the main thread becomes one of the participating threads, and the workload is divided among all the threads, including the main thread. Each thread executes a portion of the loop iterations concurrently.

It's important to note that the behavior of the main thread within the parallel region can differ depending on the OpenMP construct used. For example, in a parallel region with `#pragma omp parallel for`, the main thread participates in the loop iterations just like any other thread. However, in other constructs like `#pragma omp single`, the main thread may have a specific role and execute certain parts of the code exclusively.

When writing parallel code using OpenMP, it is crucial to consider thread synchronization and data sharing aspects to ensure correct and efficient parallel execution.

1.2 Data Scoping

```
private(n) //Declare a variable as private, each thread gets its own copy
shared(n) //Declare a variable as shared, all threads access the same memory location
reduction(n) //Perform a reduction operation (e.g., sum) on a variable across threads
default //Specify the default data scoping for variables in a parallel region
firstprivate //Declare a variable as private and initialize it with the value from the master thread
lastprivate //Copy the value of a variable from the last iteration of a loop to the master thread
```

1.2.1 Private

- **private** (comma-separated-list-of-var-names): Declares variables in its list to be private to each thread.
- A new object of the same type is declared once for each thread in the team.
- All references to the original object are replaced with references to the new object.
- Should be assumed to be uninitialized for each thread.

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

int main() {
    int n = 0;
    #pragma omp parallel private(n)
    {
        // Each thread will have its own copy of 'n'
        n = omp_get_thread_num();
        printf("Thread %d: n = %d\n", omp_get_thread_num(), n);
    }
    return 0;
}
```

1.2.2 Shared

- **shared** (comma-separated-list-of-var-names): Declares variables in its list to be shared among all threads in the team.
- A shared variable exists in only one memory location and all threads can read or write to that address.
- It is the programmer's responsibility to ensure that multiple threads properly access `SHARED` variables (such as via `CRITICAL` sections).

```c
#include <stdio.h>
#include <omp.h>
int main() {
    int n = 0;
    #pragma omp parallel shared(n)
    {
        // All threads will access the same memory location for 'n'
        #pragma omp critical
        {
            n += 1;
        }
        printf("Thread %d: n = %d\n", omp_get_thread_num(), n);
    }
    return 0;
}
```

1.2.3 Reduction:

- The reduction operation is performed on each private copy of the variable.
- The final result is then combined across all threads using the specified reduction operator.
- Common reduction operators include `+`, `*`, `max`, `min`, etc.

```c
#include <stdio.h>
#include <omp.h>
int main() {

    int sum = 0;

    #pragma omp parallel for reduction(+: sum)
    for (int i = 0; i < 10; i++) {
        sum += i;
    }
    printf("Sum = %d\n", sum);

    return 0;
}
```

1.2.4 default:

- `default (shared | none)`
  Allows the user to specify a default shared scope for all variables in the lexical extent of any parallel region.
- Specific variables can be exempted from the default using the `private`, `shared`, `firstprivate`, `lastprivate`, and `reduction` clauses.
- Using `none` as a default requires that the programmer explicitly scope all variables.

```c
#include <stdio.h>
#include <omp.h>
int main() {
    int num_threads = 0;
    int thread_num = 0;
    #pragma omp parallel private(thread_num) default(none)
    {
        thread_num = omp_get_thread_num();
        num_threads = omp_get_num_threads();
        printf("Thread %d out of %d threads\n", thread_num, num_threads);
    }

    return 0;
}
/*
In this example, the default(none) directive is used to specify that the data scoping
for variables should not have a default behavior. All variables used within the
parallel region must be explicitly scoped. The private(thread_num) directive is used
to declare the variable thread_num as private, ensuring that each thread
has its own copy.
The variables num_threads and thread_num are then printed within the parallel region.
*/
```

1.2.5 firstprivate

- Combines the behavior of the `PRIVATE` clause with automatic initialization of the variables in its list.
- Listed variables are initialized according to the value of their original objects prior to entry into the parallel or work-sharing construct.

```c
#include <stdio.h>
#include <omp.h>
int main() {
    // Set the number of threads used in a parallel region
    omp_set_num_threads(4);
    int x=55;
    printf("Before Shared Region=%i\n", x);
#pragma omp parallel firstprivate(x)
    {
        x++;
        printf("Thread %i   x=%i\n", omp_get_thread_num(), x);
    }
    printf("After Shared Region=%i\n", x);
    return 0;
}
/*output:
Before Shared Region = 55
Thread 0   x = 56
Thread 2   x = 56
Thread 1   x = 56
Thread 3   x = 56
After Shared Region = 55*/
```

1.2.6 lastprivate

- Combines the behavior of the PRIVATE clause with a copy from the last loop iteration or section to the original variable object.
- The value copied back into the original variable object is obtained from the last (sequentially) iteration or section of the enclosing construct.

```c
#include <stdio.h>
#include <omp.h>
int main() {
    // Set the number of threads used in a parallel region
    omp_set_num_threads(4);
    int x = 0;
#pragma omp parallel for lastprivate(x)
    for (int i = 0; i < 4; i++) {
        x = i;
        printf("Thread %d: x = %d\n", omp_get_thread_num(), x);
    }

    printf("After parallel region: x = %d\n", x);
    return 0;
}
/*output:
Thread 0: x = 0
Thread 2: x = 2
Thread 1: x = 1
Thread 3: x = 3
After parallel region: x = 3*/
```

1.3 Loop Constructs

```
#pragma omp for //Distribute loop iterations among threads in a parallel region
#pragma omp parallel for //Distribute loop iterations among threads in a parallel region
```

1.3.1 pragma omp for

This directive is used to distribute loop iterations among threads in a parallel region. It assumes that you have already defined a parallel region using `pragma omp parallel`, which creates a team of threads. When you use `pragma omp for`, it distributes the loop iterations among the threads in the team, allowing multiple threads to execute the loop in parallel. Each thread takes a portion of the loop iterations to execute independently.

```c
#include <stdio.h>
#include <omp.h>
int main() {
    int sum = 0;
    #pragma omp parallel
    {
        #pragma omp for reduction(+: sum)
        for (int i = 0; i < 10; i++) {
            sum += i;
        }
    }
    printf("Sum = %d\n", sum);
    return 0;
}
/*
In this example, the #pragma omp for directive is used within a parallel region to
distribute the loop iterations among threads. The reduction(+: sum) clause is added
to perform a reduction operation (summing) on the variable sum across threads.
The parallel region is explicitly defined using #pragma omp parallel, and within
that region, the loop iterations are distributed among the threads using
#pragma omp for. The partial results are combined using the reduction
operation to obtain the final sum.
*/
```

1.3.2 pragma omp parallel for

This directive combines the functionality of both `pragma omp parallel` and `pragma omp for`. It creates a parallel region and distributes the loop iterations among the threads in that region. In other words, it does two things at once: creates a team of threads and parallelizes the loop. This directive is useful when you want to parallelize a loop and create a parallel region in a single step.

```c
#include <stdio.h>
#include <omp.h>
int main() {
    int sum = 0;
    #pragma omp parallel for reduction(+: sum)
    for (int i = 0; i < 10; i++) {
        sum += i;
    }
    printf("Sum = %d\n", sum);
    return 0;
}
/*
In this example, the #pragma omp parallel for directive is used to distribute the loop
iterations among threads in a parallel region. */
```

1.4 Scheduling Examples

1.4.1 Static Scheduling with Chunk Size

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

int main() {
    int i;

    #pragma omp parallel for schedule(static, 4)
    for (i = 0; i < 16; i++) {
        printf("Thread %d executes iteration %d\n", omp_get_thread_num(), i);
    }
    return 0;
}
```

In this example, the iterations of the loop will be divided into chunks of size 4. Each thread will be assigned a chunk of iterations to execute. The output may vary depending on how the iterations are divided among the threads.

1.4.2 Dynamic Scheduling with Chunk Size

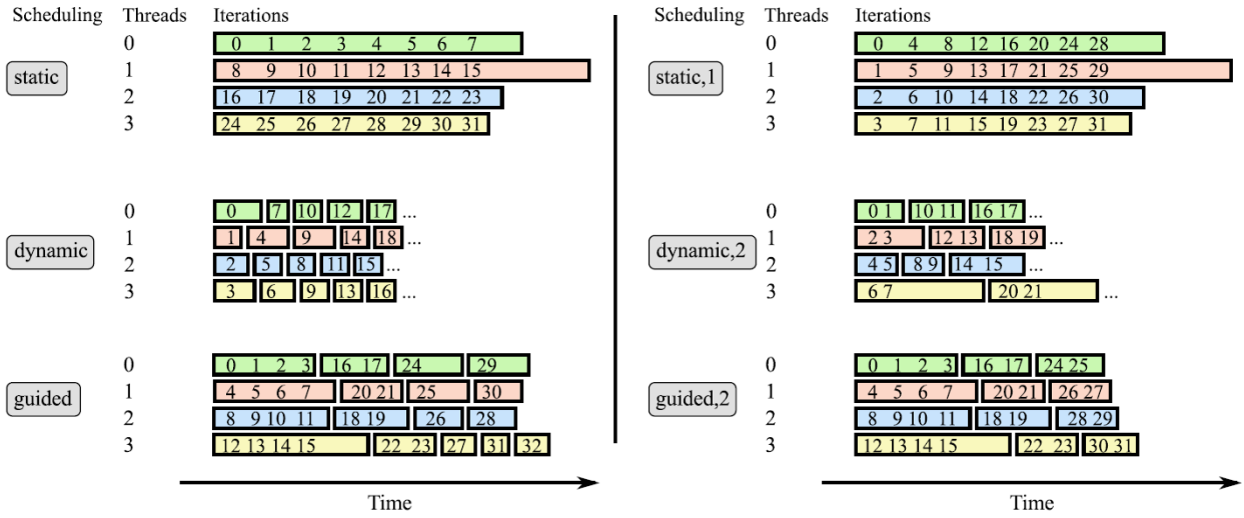
```
#include <omp.h>
#include <stdio.h>
int main() {

    int i;

    #pragma omp parallel for schedule(dynamic, 4)
    for (i = 0; i < 16; i++) {
        printf("Thread %d executes iteration %d\n", omp_get_thread_num(), i);
    }
    return 0;
}
```

In this example, the iterations will be dynamically assigned to the threads in chunks of size 4. Each thread will grab a chunk of iterations to execute. Dynamic scheduling can be useful when the workload is unbalanced.

1.4.3 Guided Scheduling with Chunk Size

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

int main() {
    int i;

    #pragma omp parallel for schedule(guided, 4)
    for (i = 0; i < 16; i++) {
        printf("Thread %d executes iteration %d\n", omp_get_thread_num(), i);
    }

    return 0;
}
```

In this example, the guided scheduling assigns larger chunks of iterations initially and gradually reduces the chunk size as the iterations progress. It is useful when the computation has increasing iteration lengths, such as the prime sieve test.



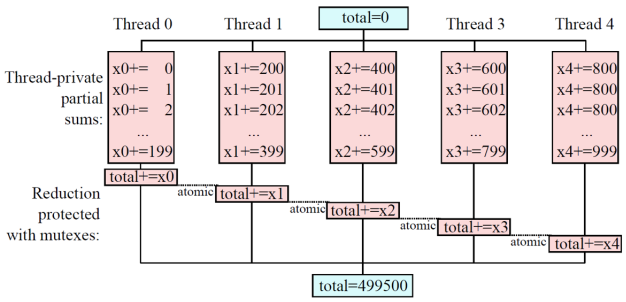
1.5 Race Conditions

```
#pragma omp atomic //Specify that an operation is atomic and should not be subject to race conditions
#pragma omp critical //Specify a critical section of code that can only be executed by 1 thread at a time
```

1.5.1 atomic

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

int main() {
    int counter = 0;

    #pragma omp parallel for
    for (int i = 0; i < 10; i++) {
        #pragma omp atomic
        counter++;
    }

    printf("Counter = %d\n", counter);

    return 0;
}
```

```
int total = 0;
#pragma omp parallel
{
  int total_thr = 0;
#pragma omp for
  for (int i=0; i<n; i++)
    total_thr += i;

#pragma omp atomic
  total += total_thr;

}
```



1.5.2 critical

```
#include <stdio.h>
#include <omp.h>
int main() {
    int counter = 0;
    #pragma omp parallel for
    for (int i = 0; i < 10; i++) {
        #pragma omp critical
        {
            counter++;
        }
    }
    printf("Counter = %d\n", counter);
    return 0;
}
```

1.6 Explicit Barrier

```
#pragma omp barrier //Explicitly synchronize all threads at a certain point in the code
```

```
#include <stdio.h>
#include <omp.h>
int main() {
#pragma omp parallel
    {
        printf("Before barrier - Thread %d\n", omp_get_thread_num());
        #pragma omp barrier
        printf("After barrier - Thread %d\n", omp_get_thread_num());
    }
    return 0;
}
```

1.7 Sections

```
#pragma omp sections // Divide the work among sections, each section executed by a thread
```

```
#include <stdio.h>
#include <omp.h>
int main() {
#pragma omp parallel sections num_threads(4)
    {
#pragma omp section
        {
            printf("Section 1 - Thread %d\n", omp_get_thread_num());
        }
#pragma omp section
        {
            printf("Section 2 - Thread %d\n", omp_get_thread_num());
        }
#pragma omp section
        {
            printf("Section 3 - Thread %d\n", omp_get_thread_num());
        }
    }
    return 0;
}//Section 1 - Thread 0 | Section 2 - Thread 1 | Section 3 - Thread 2
```

2 Exercises

2.1 PI Calculation

Serial PI Program:

```
#include<stdio.h>
static long num_steps = 100000000;
double step;
int main ()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
    printf("pi = %lf", pi);
}
```

Create a parallel version of the pi program.

Answer:

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

static long num_steps = 100000000;
double step;

int main() {
    int i;
    double x, pi, sum = 0.0;
    step = 1.0 / (double)num_steps;

    omp_set_num_threads(10);

#pragma omp parallel private(x) reduction(+:sum)
    {
#pragma omp for
        for (i = 0; i < num_steps; i++) {
            x = (i + 0.5) * step;
            sum += 4.0 / (1.0 + x * x);
        }
    }

    pi = step * sum;
    printf("pi = %lf\n", pi);

    return 0;
}
```

2.2 Array Operations

Define an array using C language, and then implement the following operations in serial code:

- Fill the array with user values.
- Search for the maximum value in the array.
- Search for a specific value in the array (value entered by the user). Print 1 if it exists, 0 otherwise.
- Count the number of occurrences of a specific value in the array (value entered by the user).
- Calculate the multiplication of all even numbers in the array.
- Compare two arrays. Display 1 if they are identical, 0 otherwise.
- Calculate the average of the values in the array.

Parallelize your code and measure the processing time for both the serial and parallel implementations. Vary the number of threads and consider different schedule modes in the for loop.

Now, perform the following tasks in parallel using 4 threads:

- Repeatedly find the maximum element in the array for each thread individually.
- At the end, the master thread must verify if all the maximum values are equal. (Hint: Use sections)

Answer:

```
#include <iostream>
#include <vector>
#include <omp.h>
using namespace std;

void fillArray(vector<int>& array)
{
    cout << "Enter " << array.size() << " values:\n";
    for (int i = 0; i < array.size(); i++)
    {
        cout << "Enter Value of index" << i << ": ";
        cin >> array[i];
    }
}
int findMax(const vector<int>& array)
{
    int max = array[0];
    int local_max = max;
#pragma omp parallel firstprivate(local_max)
    {
#pragma omp for
        for (int i = 1; i < array.size(); i++)
        {
            if (array[i] > local_max)
                local_max = array[i];
        }
#pragma omp critical
        {
            if (local_max > max)
                max = local_max;
        }
    }
    return max;
}
int searchValue(const vector<int>& array, int val)
{
    int found = 0;
#pragma omp parallel for reduction(|:found)
    for (int i = 0; i < array.size(); i++)
```

```
    {
        if (array[i] == val)
        {
            found = 1;
        }
    }
    return found;
}
int countOccurrences(const vector<int>& array, int val) {
    int count = 0;
#pragma omp parallel for reduction(+:count)
    for (int i = 0; i < array.size(); i++) {
        if (array[i] == val) {
            count++;
        }
    }
    return count;
}
int multiplyEvenNumbers(const vector<int>& array) {
    int result = 1;
#pragma omp parallel for reduction(*:result)
    for (int i = 0; i < array.size(); i++) {
        if (array[i] % 2 == 0) {
            result *= array[i];
        }
    }
    return result;
}
int compareArrays(const vector<int>& array1, const vector<int>& array2) {
    int equal = 1;
#pragma omp parallel for
    for (int i = 0; i < array1.size(); i++) {
        if (array1[i] != array2[i]) {
            equal = 0;
        }
    }
    return equal;
}
double calculateAverage(const vector<int>& array) {
    double sum = 0.0;
#pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < array.size(); i++) {
        sum += array[i];
    }
    return sum / array.size();
}

int main() {
    const int ARRAY_SIZE = 10;
    vector<int> array(ARRAY_SIZE);
    int max, val, count, result, equal;
    double average;

    fillArray(array);

    double start_time = omp_get_wtime();

    max = findMax(array);
    cout << "Maximum value: " << max << endl;

    cout << "Enter a value to search: ";
    cin >> val;
    int found = searchValue(array, val);
    cout << "Search result: " << found << endl;

    count = countOccurrences(array, val);
    cout << "Number of occurrences: " << count << endl;
```

```
107
108     result = multiplyEvenNumbers(array);
109     cout << "Multiplication of even numbers: " << result << endl;
110
111     vector<int> array2(ARRAY_SIZE);
112     fillArray(array2);
113     equal = compareArrays(array, array2);
114     cout << "Array comparison result: " << equal << endl;
115
116     average = calculateAverage(array);
117     cout << "Average: " << average << endl;
118
119     double end_time = omp_get_wtime();
120     cout << "Serial execution time: " << end_time - start_time << " seconds" << endl;
121
122     // Parallel code
123     cout << "\nParallel code:\n";
124
125     start_time = omp_get_wtime();
126
127 #pragma omp parallel sections
128     {
129 #pragma omp section
130         {
131             max = findMax(array);
132             cout << "Maximum value: " << max << endl;
133         }
134
135 #pragma omp section
136         {
137             int local_found = searchValue(array, val);
138 #pragma omp critical
139             found = local_found;
140         }
141 #pragma omp section
142         {
143             count = countOccurrences(array, val);
144 #pragma omp critical
145             cout << "Number of occurrences: " << count << endl;
146         }
147     }
148     result = multiplyEvenNumbers(array);
149     cout << "Multiplication of even numbers: " << result << endl;
150
151 #pragma omp parallel sections
152     {
153 #pragma omp section
154         {
155             equal = compareArrays(array, array2);
156 #pragma omp critical
157             cout << "Array comparison result: " << equal << endl;
158         }
159 #pragma omp section
160         {
161             average = calculateAverage(array);
162 #pragma omp critical
163             cout << "Average: " << average << endl;
164         }
165     }
166     end_time = omp_get_wtime();
167     cout << "Parallel execution time: " << end_time - start_time << " seconds" << endl;
168     return 0;
169 }
```

2.3 Square Elements of an Array

Write an application that calculates for each array element the square of their index and stores it as the element value. Of course, these calculations must be done in parallel

Answer:

```cpp
#include <iostream>
#include <omp.h>

int main() {
    const int size = 10;
    int array[size];

    #pragma omp parallel for
    for (int i = 0; i < size; ++i) {
        array[i] = i * i;
    }

    std::cout << "Array elements after squaring their indices:\n";
    for (int i = 0; i < size; ++i) {
        std::cout << array[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```