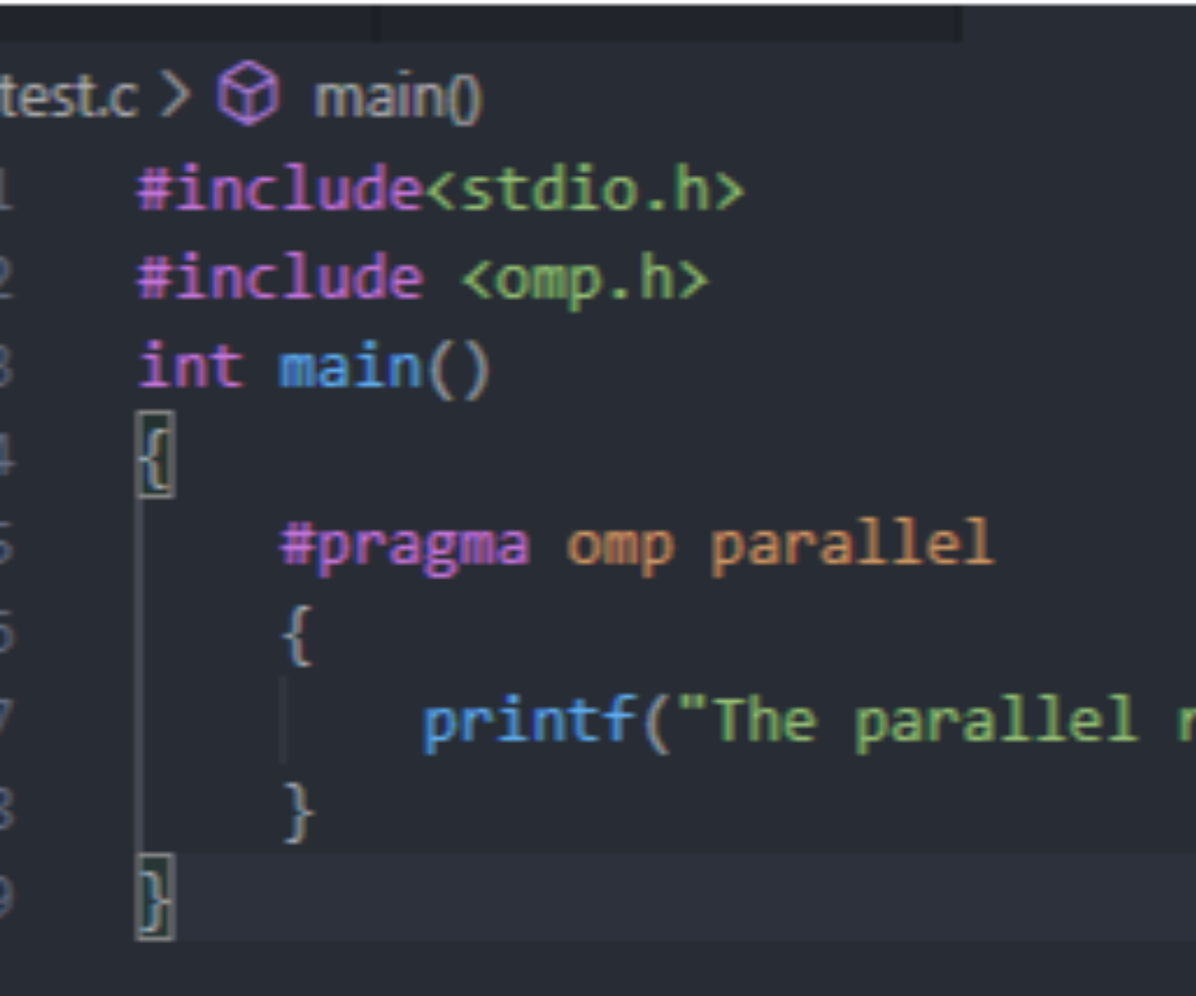

C HPC OpenMP

OpenMP - Scheduling(static, dynamic, guided, runtime, auto)

High Performance Computing

Posted by Yiling on July 15, 2020

What is Scheduling in OpenMP

Scheduling is a method in OpenMP to distribute iterations to different threads in `for` loop.

The basic form of OpenMP scheduling is

```
#pragma omp parallel for schedule(scheduling-type) for(conditions){
    do something
}
```

Of course you can use `#pragma omp parallel for` directly without scheduling, it is equal to `#pragma omp parallel for schedule(static,1)` [1]

If you run

```
int main()
{
#pragma omp parallel for schedule(static,1)     for (int i = 0; i < 20; i++)
        {
                printf("Thread %d is running number %d\n", omp_get_thread_num(), i);
        }
        return 0;
}
```

and

```
int main()
{
#pragma omp parallel for        for (int i = 0; i < 20; i++)
        {
                printf("Thread %d is running number %d\n", omp_get_thread_num(), i);
        }
        return 0;
}
```

The result stays similar. 20 tasks distributes on 12 threads on my 6-core cpu machine (thread_number = core_number * 2) equally, order to print the result is quite random, but not a big issue(if you run the same code for multiple times, the printed might be different, too)

Result 1:

```
Thread 5 is running number 5
Thread 5 is running number 17
Thread 1 is running number 1
Thread 1 is running number 13
Thread 3 is running number 3
Thread 3 is running number 15
Thread 6 is running number 6
Thread 6 is running number 18
Thread 0 is running number 0
Thread 0 is running number 12
Thread 9 is running number 9
Thread 4 is running number 4
Thread 4 is running number 16
Thread 2 is running number 2
Thread 2 is running number 14
Thread 7 is running number 7
Thread 7 is running number 19
Thread 10 is running number 10
Thread 11 is running number 11
Thread 8 is running number 8
```

Result 2:

```
Thread 4 is running number 8
Thread 4 is running number 9
Thread 1 is running number 2
Thread 1 is running number 3
Thread 0 is running number 0
Thread 0 is running number 1
Thread 6 is running number 12
Thread 6 is running number 13
Thread 8 is running number 16
Thread 9 is running number 17
Thread 10 is running number 18
Thread 11 is running number 19
Thread 2 is running number 4
Thread 2 is running number 5
Thread 5 is running number 10
Thread 5 is running number 11
Thread 3 is running number 6
Thread 3 is running number 7
Thread 7 is running number 14
Thread 7 is running number 15
```

Static

```
#pragma omp parallel for schedule(static,chunk-size)
```

If you do not specify `chunk-size` variable, OpenMP will divides iterations into chunks that are approximately equal in size and it distributes chunks to threads **in order(Notice that is why static method different from others**.In the `for` loop we discussed before, under 12-thread condition, each thread will treat 1-2 iterations; if you only use 4 threads, each thread will treat 5 iterations.

Result after using `#pragma omp parallel for schedule(static)` (If you do not specify `chunk-size`, the default value is 1)

```
Thread 0 is running number 0
Thread 0 is running number 1
Thread 6 is running number 12
Thread 6 is running number 13
Thread 8 is running number 16
Thread 3 is running number 6
Thread 3 is running number 7
Thread 2 is running number 4
Thread 2 is running number 5
Thread 9 is running number 17
Thread 10 is running number 18
Thread 11 is running number 19
Thread 5 is running number 10
Thread 5 is running number 11
Thread 1 is running number 2
Thread 1 is running number 3
Thread 4 is running number 8
Thread 4 is running number 9
Thread 7 is running number 14
Thread 7 is running number 15
```

If you specify `chunk-size` variable, the iterations will be divide `into iter_size / chunk_size` chunks.

Notice: iter_size is 20 in this example, because for loop ranges from 0 to 20(not include 20 itself) here

In

```
int main()
{
#pragma omp parallel for schedule(static, 3)     for (int i = 0; i < 20; i++)
        {
                printf("Thread %d is running number %d\n", omp_get_thread_num(), i);
        }
        return 0;
}
```

20 iterations will be divided into 7 chunks(6 with 3 iters, 1 with 2 iters), the result is:

```
Thread 5 is running number 15
Thread 5 is running number 16
Thread 5 is running number 17
Thread 2 is running number 6
Thread 2 is running number 7
Thread 2 is running number 8
Thread 6 is running number 18
Thread 6 is running number 19
Thread 1 is running number 3
Thread 1 is running number 4
Thread 1 is running number 5
Thread 3 is running number 9
Thread 3 is running number 10
Thread 3 is running number 11
Thread 4 is running number 12
Thread 4 is running number 13
Thread 0 is running number 0
Thread 0 is running number 1
Thread 0 is running number 2
Thread 4 is running number 14
```

It is clear that the cpu only uses thread 0, 1, 2, 3, 4, 5, 6 here

But what if `iter_size` / `chunk_size` is larger than the number of threads in your computer, or number of threads you specified in `omp_set_num_threads(thread_num)` ?

The following example how OpenMP works under this kind of condition.

```
int main()
{
        omp_set_num_threads(4);
#pragma omp parallel for schedule(static, 3)     for (int i = 0; i < 20; i++)
        {
                printf("Thread %d is running number %d\n", omp_get_thread_num(), i);
        }
        return 0;
}
```

Result:

```
Thread 1 is running number 3
Thread 1 is running number 4
Thread 1 is running number 5
Thread 1 is running number 15
Thread 1 is running number 16
Thread 1 is running number 17
Thread 3 is running number 9
Thread 3 is running number 10
Thread 3 is running number 11
Thread 0 is running number 0
Thread 0 is running number 1
Thread 0 is running number 2
Thread 0 is running number 12
Thread 0 is running number 13
Thread 0 is running number 14
Thread 2 is running number 6
Thread 2 is running number 7
Thread 2 is running number 8
Thread 2 is running number 18
Thread 2 is running number 19
```

OpenMP will still split task into 7 chunks, but distributes the chunks to threads **in a circular order**, like the following figure shows

Dynamic

```
#pragma omp parallel for schedule(dynamic,chunk-size)
```

OpenMP will still split task into `iter_size` / `chunk_size` chunks, but distribute trunks to threads dynamically without any specific order.

If you run

```
int main()
{
#pragma omp parallel for schedule(dynamic, 1)   for (int i = 0; i < 20; i++)
        {
                printf("Thread %d is running number %d\n", omp_get_thread_num(), i);
        }
        return 0;
}
```

`#pragma omp parallel for schedule(dynamic, 1` is equivalent to `#pragma omp parallel for schedule(dynamic)`

Result:

```
Thread 1 is running number 2
Thread 1 is running number 7
Thread 1 is running number 9
Thread 1 is running number 10
Thread 1 is running number 11
Thread 1 is running number 13
Thread 1 is running number 14
Thread 1 is running number 15
Thread 1 is running number 17
Thread 1 is running number 19
Thread 3 is running number 0
Thread 0 is running number 4
Thread 8 is running number 12
Thread 4 is running number 3
Thread 6 is running number 6
Thread 9 is running number 16
Thread 5 is running number 1
Thread 7 is running number 8
Thread 10 is running number 18
Thread 2 is running number 5
```

You can see that thread 1 took on 10 iters while others took only 0-1.

Comparing with `static` Method:

Pros: The dynamic scheduling type is appropriate when the iterations require different computational costs. This means that the iterations are not as balance as static method between each other.

Cons: The dynamic scheduling type has higher overhead then the static scheduling type because it dynamically distributes the iterations during the runtime.[1]

Guided

```
#pragma omp parallel for schedule(guided,chunk-size)
```

Chunk size is dynamic while using guided method, the size of a chunk is proportional to the number of unassigned iterations divided by the number of the threads, and the size will be decreased to `chunk-size` (but the last chunk could be smaller than `chunk-size`)

Use a 4-thread structure to see what will happen in a 20-iter `for` loop after applying guided method:

```
int main()
{
        omp_set_num_threads(4);
#pragma omp parallel for schedule(guided, 3)    for (int i = 0; i < 20; i++)
        {
                printf("Thread %d is running number %d\n", omp_get_thread_num(), i);
        }
        return 0;
}
```

Result:

```
Thread 1 is running number 5
Thread 1 is running number 6
Thread 1 is running number 7
Thread 1 is running number 8
Thread 1 is running number 15
Thread 1 is running number 16
Thread 1 is running number 17
Thread 1 is running number 18
Thread 1 is running number 19
Thread 3 is running number 12
Thread 3 is running number 13
Thread 3 is running number 14
Thread 0 is running number 0
Thread 0 is running number 1
Thread 2 is running number 9
Thread 2 is running number 10
Thread 2 is running number 11
Thread 0 is running number 2
Thread 0 is running number 3
Thread 0 is running number 4
```

Runtime

Depend on environment variable `OMP_SCHEDULE` we set in command line.

Auto

Will delegates the decision of the scheduling to the compiler and/or runtime system. That means, scheduling will be decided automatically by your machine.

Reference

[1] Jaka's Corner

Some concepts from:

OpenMP并行构造的schedule子句详解

Programming Parallel Computers

PREVIOUS
ENABLE OPENMP IN VISUAL STUDIO

NEXT
OPENMP - TASK AND SECTIONS