- ## #pragma omp task

One thread creates a task to write the tasks onto sticky notes.

All threads to execute the sticky notes.

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

using namespace std;
int main() {
       omp_set_num_threads( 2 );
#pragma omp parallel
{
#pragma omp task
  cout<<"A"<<endl;
#pragma omp task
  cout<<"B"<<endl;
}

    return 0;
}
```

Que- Why do we get 4 things printed when we only have print statements in 2 tasks?

Ans: Each of the two threads was assigned two tasks, resulting in a total of four tasks on the sticky notes

- ## #pragma omp taskwait

Causes all tasks to wait until they are completed.

If you care about order, do this:

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

using namespace std;
int main() {
       omp_set_num_threads(2);
#pragma omp parallel
{
       #pragma omp task
         cout<<"A"<<endl;
               #pragma omp taskwait
       #pragma omp task
         cout<<"B"<<endl;
               #pragma omp taskwait
}
    return 0;
}
```

Factorial using tasks

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

long factorial(int n) {
    if (n == 0 || n == 1)
     return 1;

    long result;

    #pragma omp task shared(result)

    result = n * factorial(n - 1);

    #pragma omp taskwait

    return result;
}
int main() {
    int n;
    cout << "Enter a number: ";
    cin >> n;
    long result;
    #pragma omp parallel
    {
        #pragma omp single
        result = factorial(n);
    }
    cout << "Factorial of " << n << " is " << result << endl;
    return 0;
}
```

- pragma omp parallel for collapse(2)

  Specifying the **COLLAPSE** clause allows you to parallelize multiple loops in a nest without introducing nested parallelism.

  collapse(2): The collapse clause tells OpenMP to combine the two nested loops into one large loop. The 2 in collapse(2) means that OpenMP will collapse the two innermost loops into a single loop with a single iteration space.

Example Without collapse(2)

```cpp
#include <iostream>
#include <omp.h>
using namespace std;
int main() {
    const int N = 4;
    int matrix[N][N] = {0};

    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            matrix[i][j] = i + j;
        }
    }
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

- Only the outer loop (i loop) is parallelized.
- The inner loop (j loop) runs sequentially within each thread.

Example With collapse(2)

```cpp
#include <iostream>
#include <omp.h>
using namespace std;
int main() {
    const int N = 4;
    int matrix[N][N] = {0};
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            matrix[i][j] = i + j;
        }
    }
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

What is Scheduling in OpenMP

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

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

Static

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

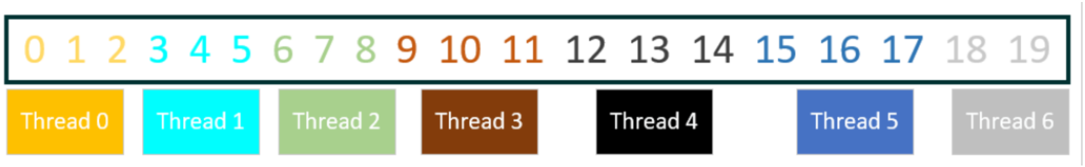
If you do not specify chunk-size variable, OpenMP will divides iterations into chunks that are approximately equal in size and it distributes chunks to threads **in order**

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

Example

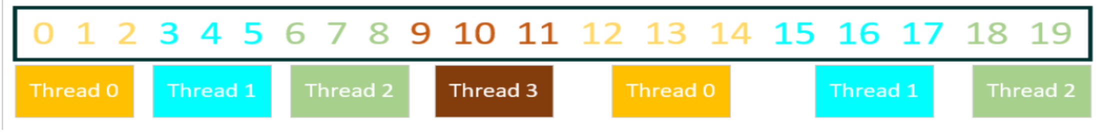
```
#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
#pragma omp parallel for schedule(static, 3)
	for (int i = 0; i < 20; i++)
	{
cout<<"Thread "<< omp_get_thread_num()<<" is 
running number \n"<< i;
	}
	return 0;
}
```

20 iterations will be divided into 7 chunks(6 with 3 iters, 1 with 2 iters).



Que: 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)?

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



Dynamic

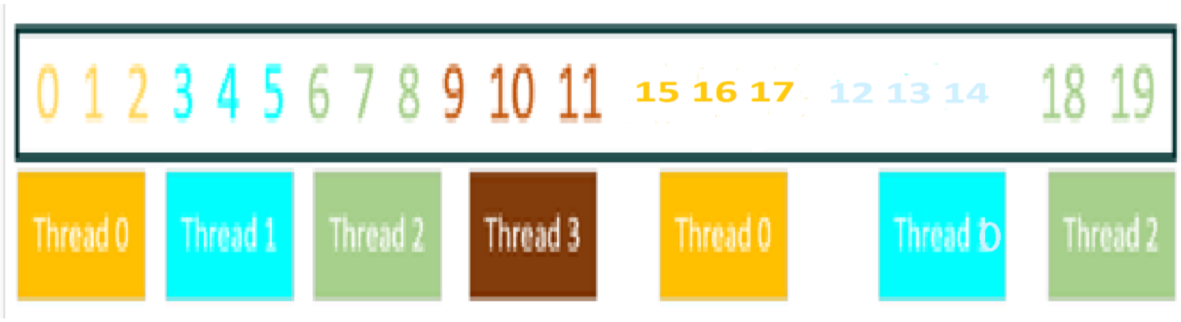
#pragma omp parallel for schedule(dynamic,chunk-size)

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

Example

```
#include <iostream>
#include <omp.h>
using namespace std;
int main()
{
#pragma omp parallel for schedule(dynamic, 3)
       for (int i = 0; i < 20; i++)
       {
cout<<"Thread "<< omp_get_thread_num()<<" is   running number \n"<< i;
       }
       return 0;
}
```

20 iterations will be divided into threads but not in order any chunk not done any thread can take it to execute.


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