Static vs Dynamic Scheduling

`static` schedule means that iterations blocks are mapped statically to the execution threads in a round-robin fashion. The nice thing with static scheduling is that OpenMP run-time guarantees that if you have two separate loops with the same number of iterations and execute them with the same number of threads using static scheduling, then each thread will receive exactly the same iteration range(s) in both parallel regions. This is quite important on NUMA systems: if you touch some memory in the first loop, it will reside on the NUMA node where the executing thread was. Then in the second loop the same thread could access the same memory location faster since it will reside on the same NUMA node.

Imagine there are two NUMA nodes: node 0 and node 1, e.g. a two-socket Intel Nehalem board with 4-core CPUs in both sockets. Then threads 0, 1, 2, and 3 will reside on node 0 and threads 4, 5, 6, and 7 will reside on node 1:

```
|             | core 0 | thread 0 |
| socket 0    | core 1 | thread 1 |
| NUMA node 0 | core 2 | thread 2 |
|             | core 3 | thread 3 |

|             | core 4 | thread 4 |
| socket 1    | core 5 | thread 5 |
| NUMA node 1 | core 6 | thread 6 |
|             | core 7 | thread 7 |
```

Each core can access memory from each NUMA node, but remote access is slower (1.5x - 1.9x slower on Intel) than local node access. You run something like this:

```
char *a = (char *)malloc(8*4096);

#pragma omp parallel for schedule(static,1) num_threads(8)
for (int i = 0; i < 8; i++)
   memset(&a[i*4096], 0, 4096);
```

4096 bytes in this case is the standard size of one memory page on Linux on x86 if huge pages are not used. This code will zero the whole 32 KiB array `a`. The `malloc()` call just reserves virtual address space but does not actually "touch" the physical memory (this is the default behaviour unless some other version of `malloc` is used, e.g. one that zeroes the memory like `calloc()` does). Now this array is contiguous but only in virtual memory. In physical memory half of it would lie in the memory attached to socket 0 and half in the memory attached to socket 1. This is so because different parts are zeroed by different threads and those threads reside on different cores and there is something called *first touch* NUMA policy which means that memory pages are allocated on the NUMA node on which the thread that first "touched" the memory page resides.

```
|             | core 0 | thread 0 | a[0]     ... a[4095]
| socket 0    | core 1 | thread 1 | a[4096]  ... a[8191]
| NUMA node 0 | core 2 | thread 2 | a[8192]  ... a[12287]
|             | core 3 | thread 3 | a[12288] ... a[16383]

|             | core 4 | thread 4 | a[16384] ... a[20479]
| socket 1    | core 5 | thread 5 | a[20480] ... a[24575]
| NUMA node 1 | core 6 | thread 6 | a[24576] ... a[28671]
|             | core 7 | thread 7 | a[28672] ... a[32768]
```

Now lets run another loop like this:

```
#pragma omp parallel for schedule(static,1) num_threads(8)
for (i = 0; i < 8; i++)
   memset(&a[i*4096], 1, 4096);
```

Each thread will access the already mapped physical memory and it will have the same mapping of thread to memory region as the one during the first loop. It means that threads will only access memory located in their local memory blocks which will be fast.

Now imagine that another scheduling scheme is used for the second loop: `schedule(static,2)`. This will "chop" iteration space into blocks of two iterations and there will be 4 such blocks in total. What will happen is that we will have the following thread to memory location mapping (through the iteration number):

```
|             | core 0 | thread 0 | a[0]     ... a[8191]  <- OK, same memory node
| socket 0    | core 1 | thread 1 | a[8192]  ... a[16383] <- OK, same memory node
| NUMA node 0 | core 2 | thread 2 | a[16384] ... a[24575] <- Not OK, remote
memory
|             | core 3 | thread 3 | a[24576] ... a[32768] <- Not OK, remote
memory

|             | core 4 | thread 4 | <idle>
| socket 1    | core 5 | thread 5 | <idle>
| NUMA node 1 | core 6 | thread 6 | <idle>
|             | core 7 | thread 7 | <idle>
```

Two bad things happen here:

- threads 4 to 7 remain idle and half of the compute capability is lost;
- threads 2 and 3 access non-local memory and it will take them about twice as much time to finish during which time threads 0 and 1 will remain idle.

So one of the advantages for using static scheduling is that it improves locality in memory access. The disadvantage is that bad choice of scheduling parameters can ruin the performance.

`dynamic` scheduling works on a "first come, first served" basis. Two runs with the same number of threads might (and most likely would) produce completely different "iteration space" -> "threads" mappings as one can easily verify:

```
$ cat dyn.c
#include <stdio.h>
#include <omp.h>

int main (void)
```

```
{
  int i;

  #pragma omp parallel num_threads(8)
  {
    #pragma omp for schedule(dynamic,1)
    for (i = 0; i < 8; i++)
      printf("[1] iter %0d, tid %0d\n", i, omp_get_thread_num());

    #pragma omp for schedule(dynamic,1)
    for (i = 0; i < 8; i++)
      printf("[2] iter %0d, tid %0d\n", i, omp_get_thread_num());
  }

  return 0;
}

$ icc -openmp -o dyn.x dyn.c

$ OMP_NUM_THREADS=8 ./dyn.x | sort
[1] iter 0, tid 2
[1] iter 1, tid 0
[1] iter 2, tid 7
[1] iter 3, tid 3
[1] iter 4, tid 4
[1] iter 5, tid 1
[1] iter 6, tid 6
[1] iter 7, tid 5
[2] iter 0, tid 0
[2] iter 1, tid 2
[2] iter 2, tid 7
[2] iter 3, tid 3
[2] iter 4, tid 6
[2] iter 5, tid 1
[2] iter 6, tid 5
[2] iter 7, tid 4
```

(same behaviour is observed when `gcc` is used instead)

If the sample code from the `static` section was run with `dynamic` scheduling instead there will be only 1/70 (1.4%) chance that the original locality would be preserved and 69/70 (98.6%) chance that remote access would occur. This fact is often overlooked and hence suboptimal performance is achieved.

There is another reason to choose between `static` and `dynamic` scheduling - workload balancing. If each iteration takes vastly different from the mean time to be completed then high work imbalance might occur in the static case. Take as an example the case where time to complete an iteration grows linearly with the iteration number. If iteration space is divided statically between two threads the second one will have three times more work than the first one and hence for 2/3 of the compute time the first thread will be idle. Dynamic schedule introduces some additional overhead but in that particular case will lead to much better workload distribution. A special kind of `dynamic` scheduling is the `guided` where smaller and smaller iteration blocks are given to each task as the work progresses.

Since precompiled code could be run on various platforms it would be nice if the end user can control the scheduling. That's why OpenMP provides the special `schedule(runtime)` clause. With `runtime` scheduling the type is taken from the content of the environment variable `OMP_SCHEDULE`. This allows to test different scheduling types without recompiling the application and also allows the end user to fine-tune for his or her platform.