Programming Parallel Computers

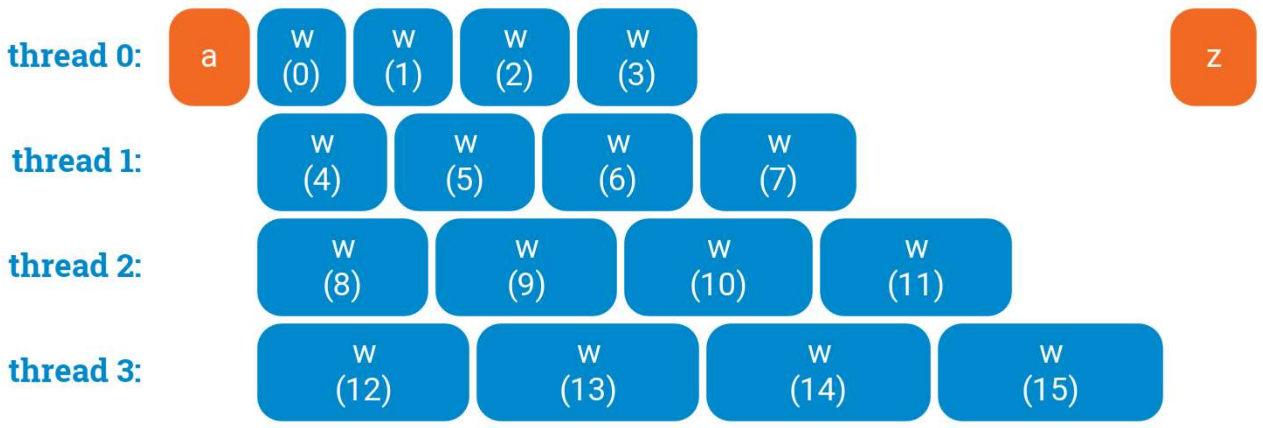
Chapter 3: Multithreading with OpenMP

OpenMP parallel for loops: scheduling

If each iteration is doing roughly the same amount of work, the standard behavior of OpenMP is usually good. For example, with 4 threads and 40 iterations, the first thread will take care of iterations 0–9, the second thread will take care of iterations 10–19, etc. This is nice especially if we are doing some memory lookups in the loop; then each thread would be doing linear reading.

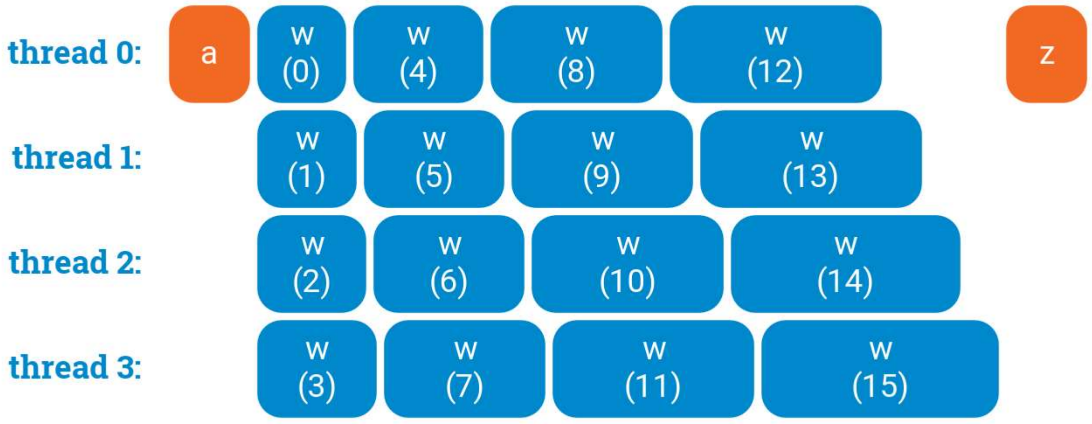
However, things are different if the amount of work that we do varies across iterations. Here we call function `w(i)` that takes time that is proportional to the value of `i`. With a normal parallel for loop, thread 0 will process all small jobs, thread 3 will process all large jobs, and hence we will need to wait a lot of time until the final thread finishes:

```
a();
#pragma omp parallel for
for (int i = 0; i < 16; ++i) {
    w(i);
}
z();
```



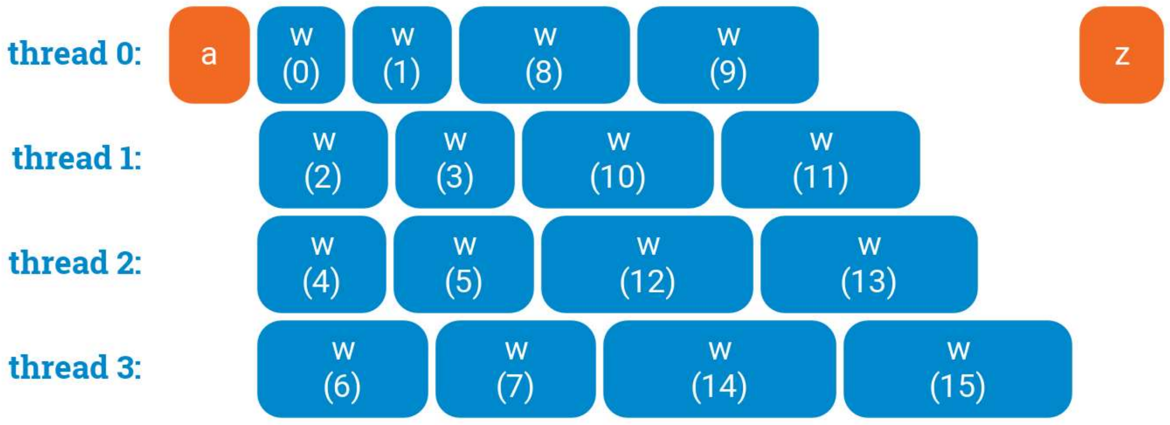
We can, however, do much better if we ask OpenMP to assign iterations to threads in a cyclic order: iteration 0 to thread 0, iteration 1 to thread 1, etc. Adding `schedule(static,1)` will do the trick:

```
a();
#pragma omp parallel for schedule(static,1)
for (int i = 0; i < 16; ++i) {
    w(i);
}
z();
```



Number “1” indicates that we assign one iteration to each thread before switching to the next thread — we use **chunks** of size 1. If we wanted to process the iterations e.g. in chunks of size 2, we could use `schedule(static,2)`, but in this case it is not useful:

```
a();
#pragma omp parallel for schedule(static,2)
for (int i = 0; i < 16; ++i) {
    w(i);
}
z();
```

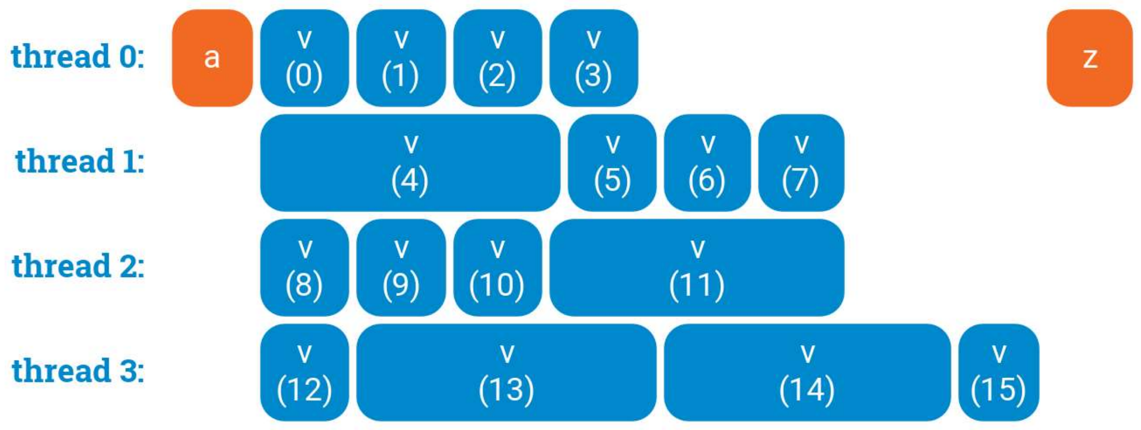


Dynamic loop scheduling

Default scheduling and static scheduling are **very efficient**: there is **no need for any communication between the threads**. When the loop starts, each thread will immediately know which iterations of the loop it will execute. The only synchronization part is at the end of the entire loop, where we just start for all threads to finish.

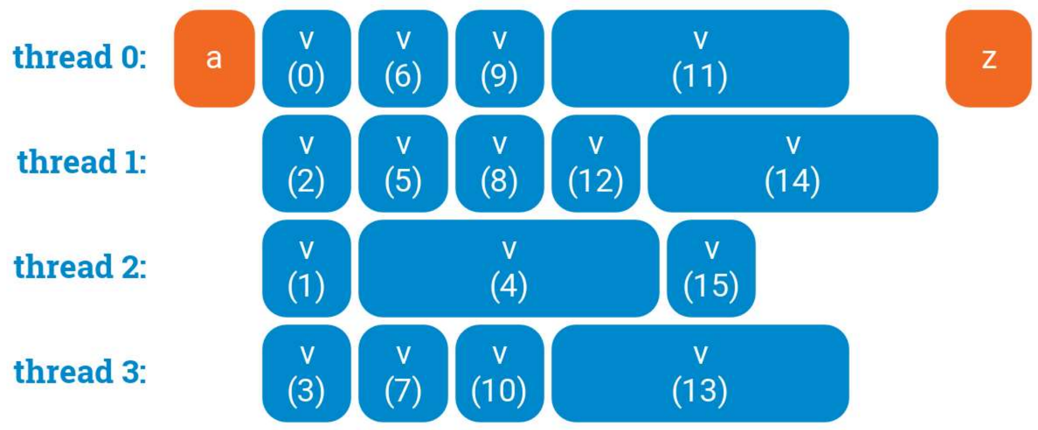
However, sometimes our workload is tricky; maybe there are some iterations that take much longer time, and for any fixed schedule we might be unlucky and some threads will run much longer:

```
a();
#pragma omp parallel for
for (int i = 0; i < 16; ++i) {
    v(i);
}
z();
```



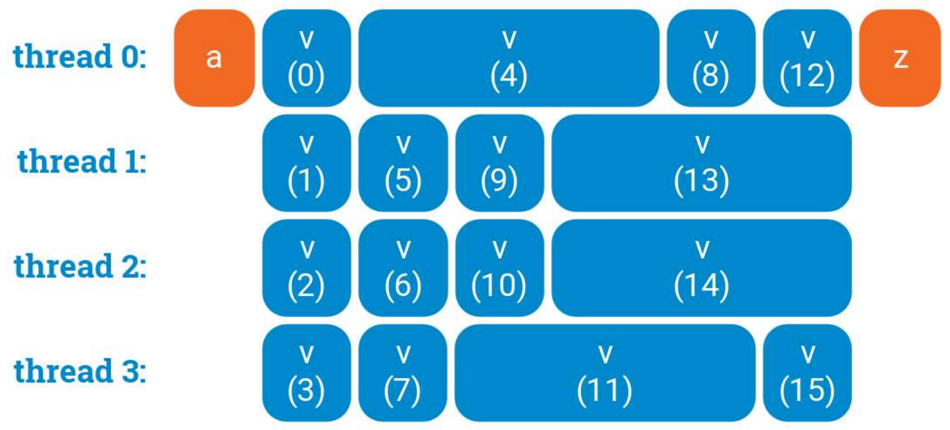
In such a case we can resort to dynamic scheduling. In dynamic scheduling, each thread will take one iteration, process it, and then see what is the next iteration that is currently not being processed by anyone. This way it will never happen that one thread finishes while other threads have still lots of work to do:

```
a();
#pragma omp parallel for schedule(dynamic,1)
for (int i = 0; i < 16; ++i) {
    v(i);
}
z();
```



However, please note that dynamic **scheduling is expensive**: there is some communication between the threads after each iteration of the loop! **Increasing the chunk size** (number “1” in the `schedule` directive) may help here to find a better trade-off between balanced workload and coordination overhead.

It is also good to note that dynamic scheduling **does not necessarily give an optimal schedule**. OpenMP cannot predict the future; it is just assigning loop iterations to threads in a greedy manner. For the above workload, we could do better e.g. as follows:


thread 0:
a
v (0)
v (4)
v (8)
v (12)
z
thread 1:
v (1)
v (5)
v (9)
v (13)
thread 2:
v (2)
v (6)
v (10)
v (14)
thread 3:
v (3)
v (7)
v (11)
v (15)