LAB 01: SINGLE PROGRAM MULTIPLE DATA (SPMD)

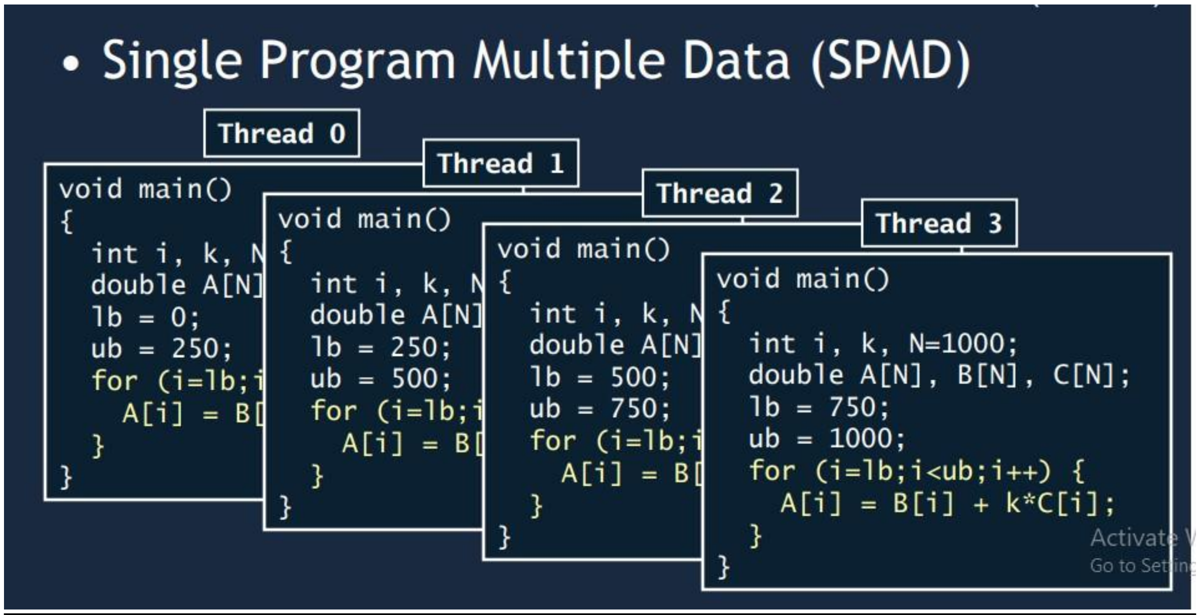
CS 3006 - Parallel and Distributed Computing

NATIONAL UNIVERSITY - FAST (FALL 2023)

Lab#01: Single Program Multiple Data (SPMD)

1. Open MP used for SPMD (Single Program Multiple Data):

Single Program Multiple Data (SPMD) is a special case of the Multiple Instruction Multiple Data model (MIMD) of Flynn's classification. In the SPMD model, a single program is executed simultaneously on multiple data elements.



• Single Program Multiple Data (SPMD)

Thread 0
```
void main()
{
  int i, k, N
  double A[N]
  lb = 0;
  ub = 250;
  for (i=lb;i
    A[i] = B[
  }
}
```

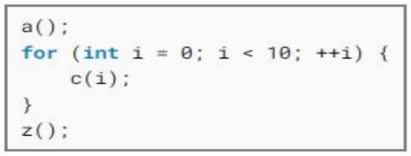
Thread 1
```
void main()
{
  int i, k, N
  double A[N]
  lb = 250;
  ub = 500;
  for (i=lb;i
    A[i] = B[
  }
}
```

Thread 2
```
void main()
{
  int i, k, N
  double A[N]
  lb = 500;
  ub = 750;
  for (i=lb;i
    A[i] = B[
  }
}
```

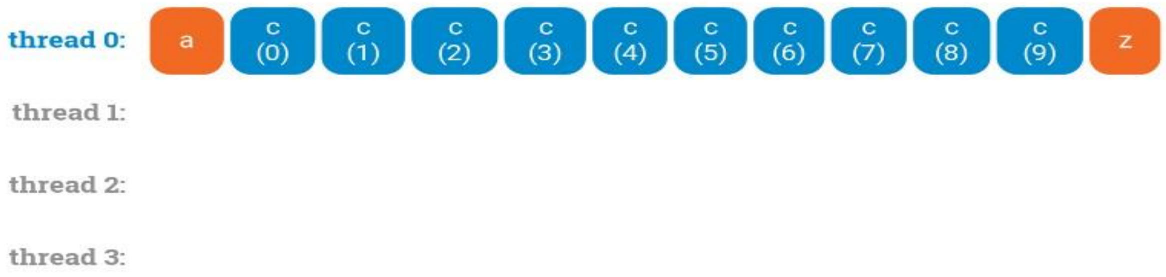
Thread 3
```
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
  lb = 750;
  ub = 1000;
  for (i=lb;i<ub;i++) {
    A[i] = B[i] + k*C[i];
  }
}
```

Parallel for Loops:

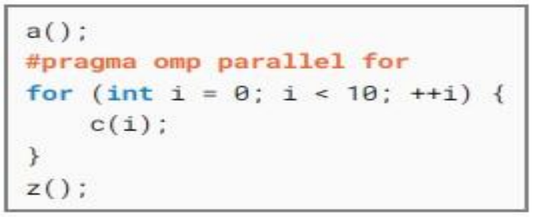
We would like to do some preprocessing operation a(), then we would like to some independent calculations c(0), c(1), …, and finally some post processing z() once all calculations have finished. In this example, each of the calculations takes roughly the same amount of time. A straightforward for-loop uses only one thread and hence only one core on a multi-core computer:



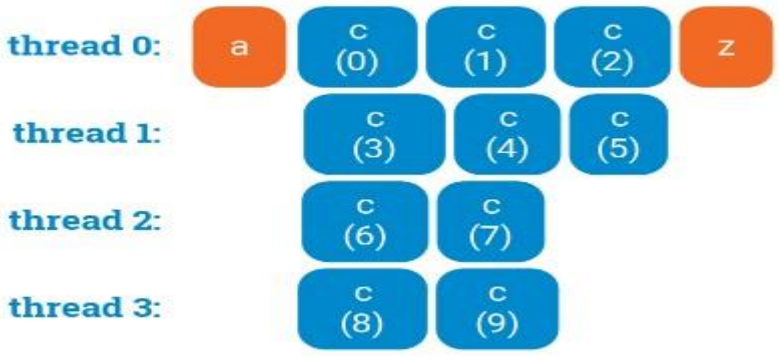
```
a();
for (int i = 0; i < 10; ++i) {
    c(i);
}
z();
```



With OpenMP parallel for loops, we can easily parallelize it so that we are making a much better use of the computer. Note that Open MP automatically waits for all threads to finish their calculations before continuing with the part that comes after the parallel for loop:

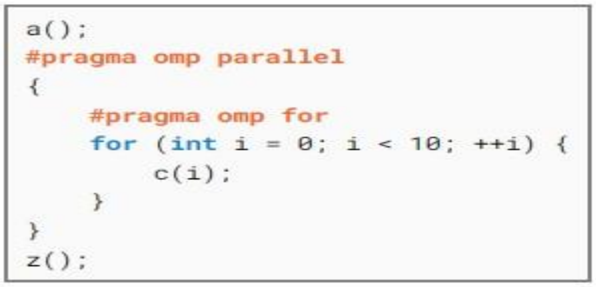


```
a();
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    c(i);
}
z();
```

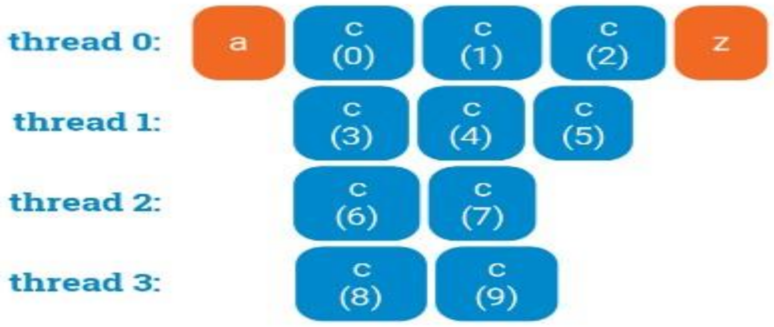


It is just a shorthand

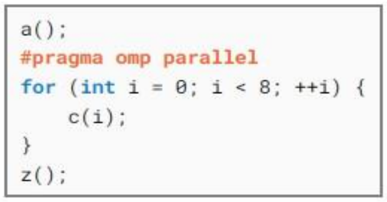
The ***omp parallel for*** directive is just a commonly-used shorthand for the combination of two directives: ***omp parallel,*** which declares that we would like to have multiple threads in this region, and ***omp for***, which asks OpenMP to assign different iterations to different threads:



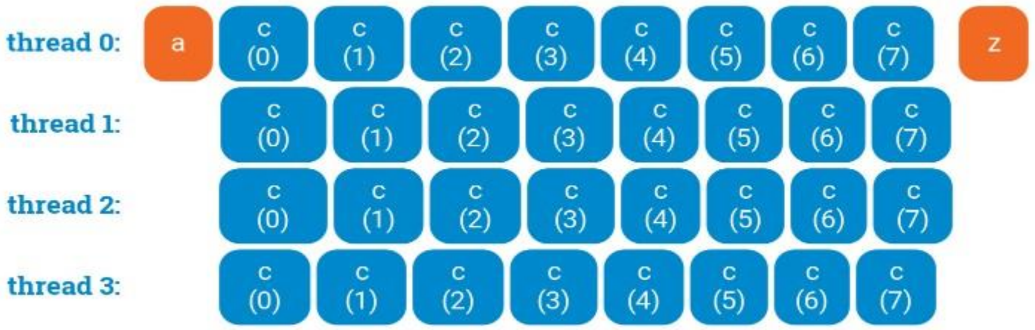
```
a();
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
}
z();
```



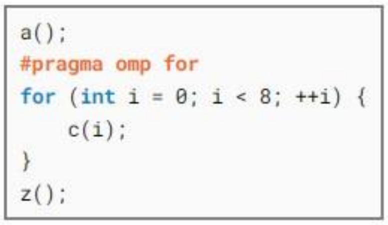
A common mistake is to just use omp parallel together with a for loop. This creates multiple threads for you, but it does not do any work-sharing — all threads simply run all iterations of the loop, which is most likely not what you want:



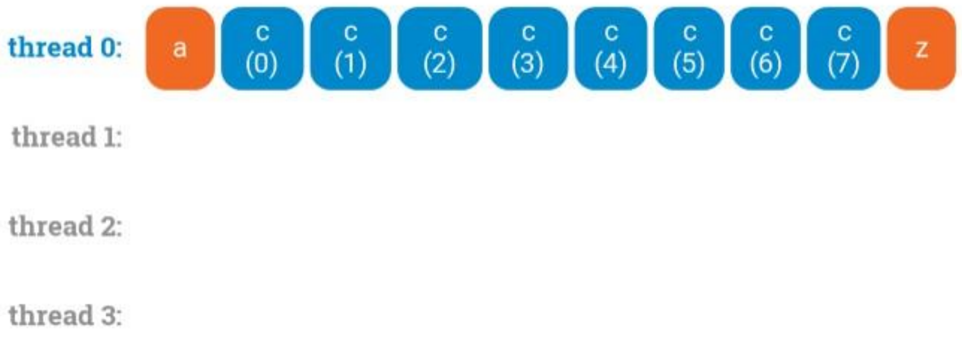
```
a();
#pragma omp parallel
for (int i = 0; i < 8; ++i) {
    c(i);
}
z();
```


thread 0: a c(0) c(1) c(2) c(3) c(4) c(5) c(6) c(7) z
thread 1: c(0) c(1) c(2) c(3) c(4) c(5) c(6) c(7)
thread 2: c(0) c(1) c(2) c(3) c(4) c(5) c(6) c(7)
thread 3: c(0) c(1) c(2) c(3) c(4) c(5) c(6) c(7)

Another common mistake is to use omp for alone outside a parallel region. It does not do anything if we do not have multiple threads available:



```
a();
#pragma omp for
for (int i = 0; i < 8; ++i) {
    c(i);
}
z();
```



Example#1: Parallelizing Array Addition using parallel for

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

#define ARRAY_SIZE 10

int main() {
    int  array1[ARRAY_SIZE];
    int  array2[ARRAY_SIZE];
    int result[ARRAY_SIZE];

    // Initialize arrays with some values
    for (int i = 0; i < ARRAY_SIZE; i++) {
        array1[i] = i;
        array2[i] = 2 * i;
    }

    // Parallelize the array addition using OpenMP
    #pragma omp parallel for
    for (int i = 0; i < ARRAY_SIZE; i++) {
        result[i] = array1[i] + array2[i];
    }

    // Print the result
    printf("Array 1: ");
    for (int i = 0; i < ARRAY_SIZE; i++) {
        printf("%d ", array1[i]);
    }
    printf("\n");

    printf("Array 2: ");
    for (int i = 0; i < ARRAY_SIZE; i++) {
        printf("%d ", array2[i]);
    }
    printf("\n");

    printf("Result: ");
    for (int i = 0; i < ARRAY_SIZE; i++) {
        printf("%d ", result[i]);
    }
    printf("\n");

    return 0;
}
```

Creating Threads:

Each thread execute a single task

–Task id is the same as thread id

• ***omp_get_thread_num()***

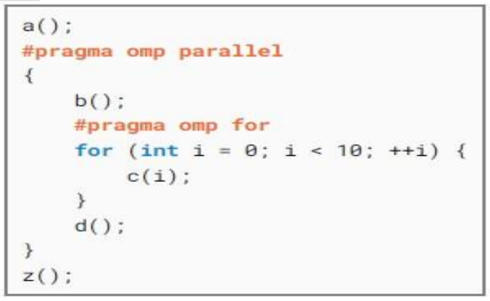
– Num_tasks is the same as total number of threads

• ***omp_get_num_threads()***

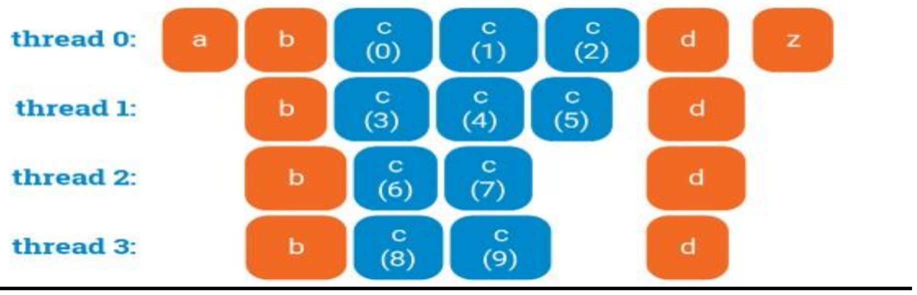
Task#1: You need to increase the array size from 1000, 3000, 5000…..10000 in Example#1. You need to create two threads, three threads and four threads to divide the task. Calculate the time increasing the size of an array. Construct a table to show your result.

Parallel for Loops: waiting

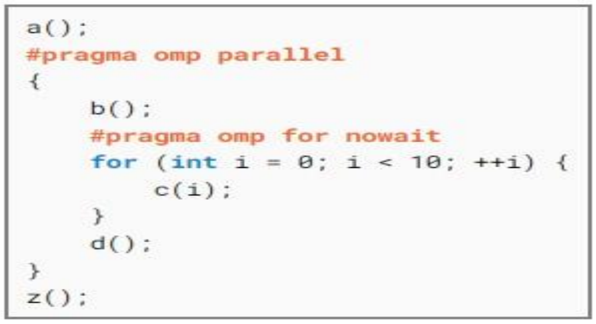
When you use a parallel region, OpenMP will automatically wait for all threads to finish before execution continues. There is also a synchronization point **after** each omp for loop; here no thread will execute d() until all threads are done with the loop:



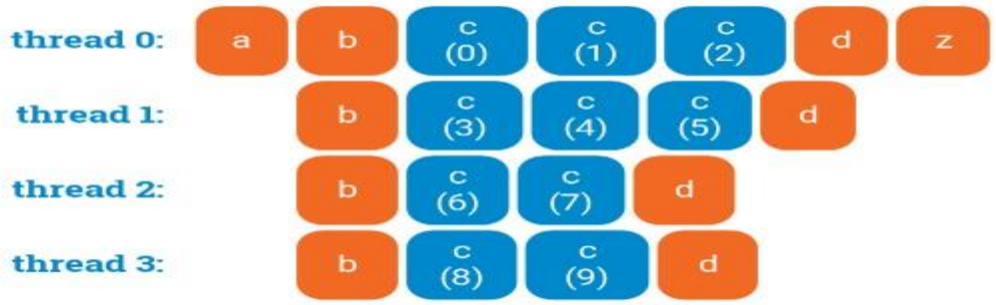
```
a();
#pragma omp parallel
{
    b();
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    d();
}
z();
```



However, if you do not need synchronization after the loop, you can disable it with nowait:



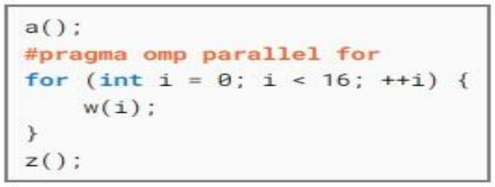
```
a();
#pragma omp parallel
{
    b();
    #pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    d();
}
z();
```



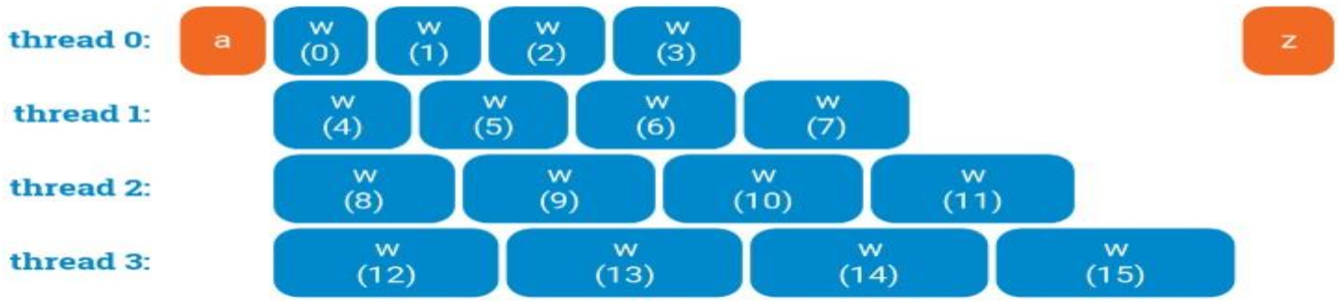
Parallel for Loops: scheduling

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.

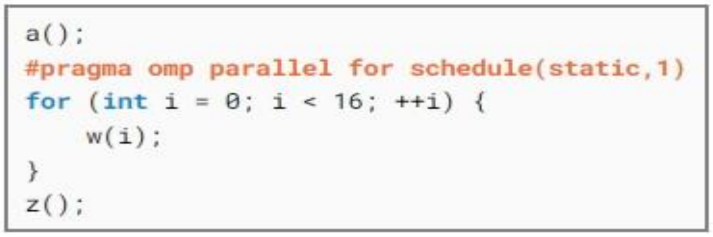
However, things are different if the amount of work that we do varies across iterations. Here we call function w(i) that takes time that is proportional to the value of i. With a normal parallel for loop, thread 0 will process all small jobs, thread 3 will process all large jobs, and hence we will need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



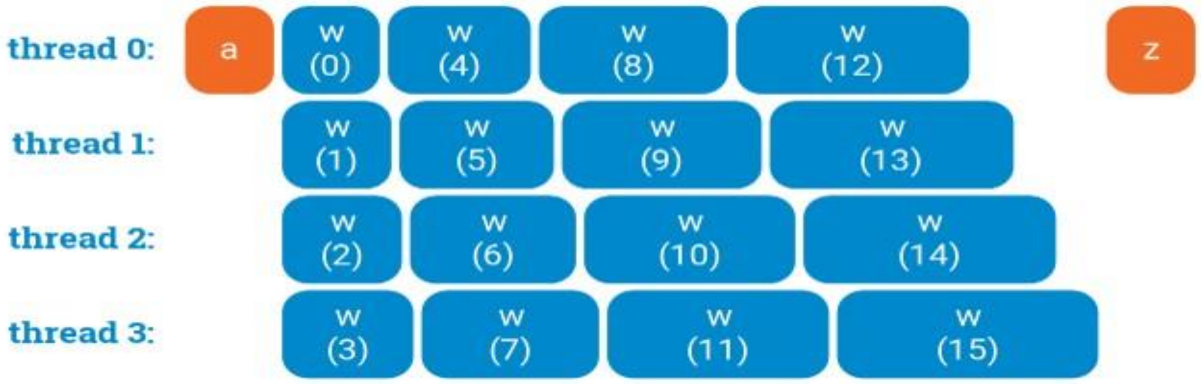
```
a();
#pragma omp parallel for
for (int i = 0; i < 16; ++i) {
    w(i);
}
z();
```



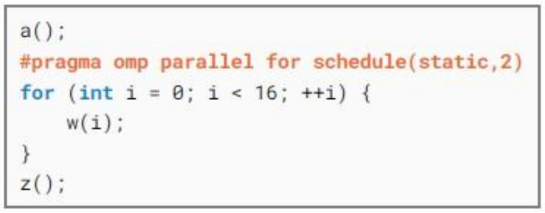
We can, however, do much better if we ask OpenMP to assign iterations to threads in a cyclic order: iteration 0 to thread 0, iteration 1 to thread 1, etc. Adding schedule(static,1) will do the trick:



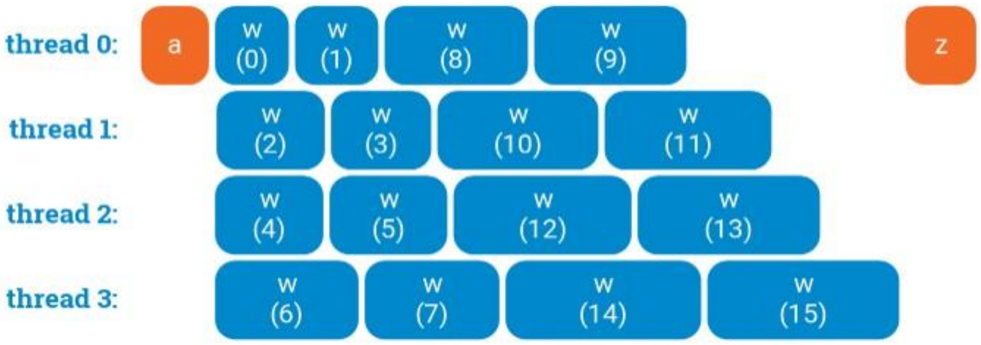
```
a();
#pragma omp parallel for schedule(static,1)
for (int i = 0; i < 16; ++i) {
    w(i);
}
z();
```



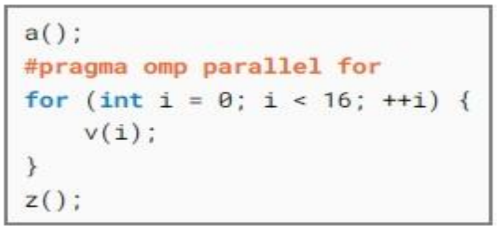
Number "1" indicates that we assign one iteration to each thread before switching to the next thread — we use **chunks** of size 1. If we wanted to process the iterations e.g. in chunks of size 2, we could use schedule(static,2), but in this case it is not useful:



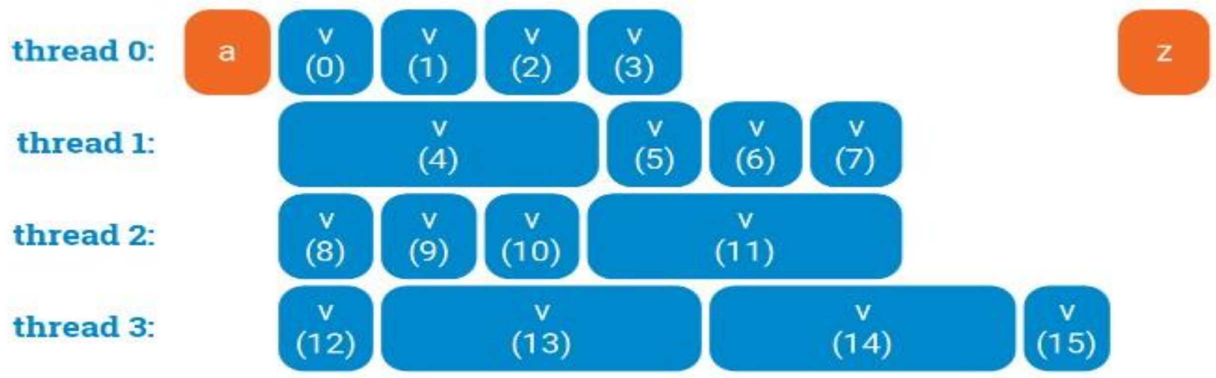
```
a();
#pragma omp parallel for schedule(static,2)
for (int i = 0; i < 16; ++i) {
    w(i);
}
z();
```



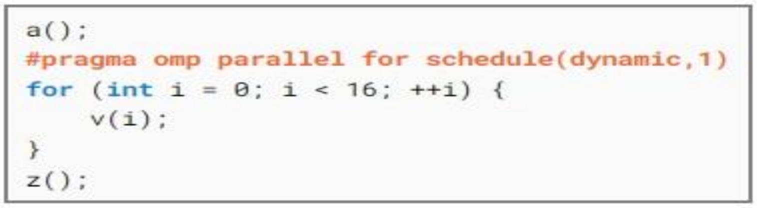
<u>Dynamic loop scheduling:</u>
Default scheduling and static scheduling are **very efficient**: there is **no need for any communication between the threads**. When the loop starts, each thread will immediately know which iterations of the loop it will execute. The only synchronization part is at the 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:



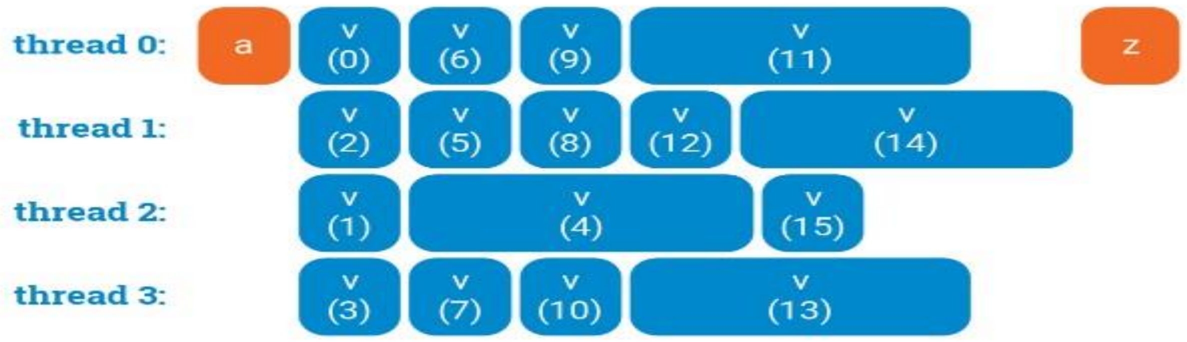
```
a();
#pragma omp parallel for
for (int i = 0; i < 16; ++i) {
    v(i);
}
z();
```



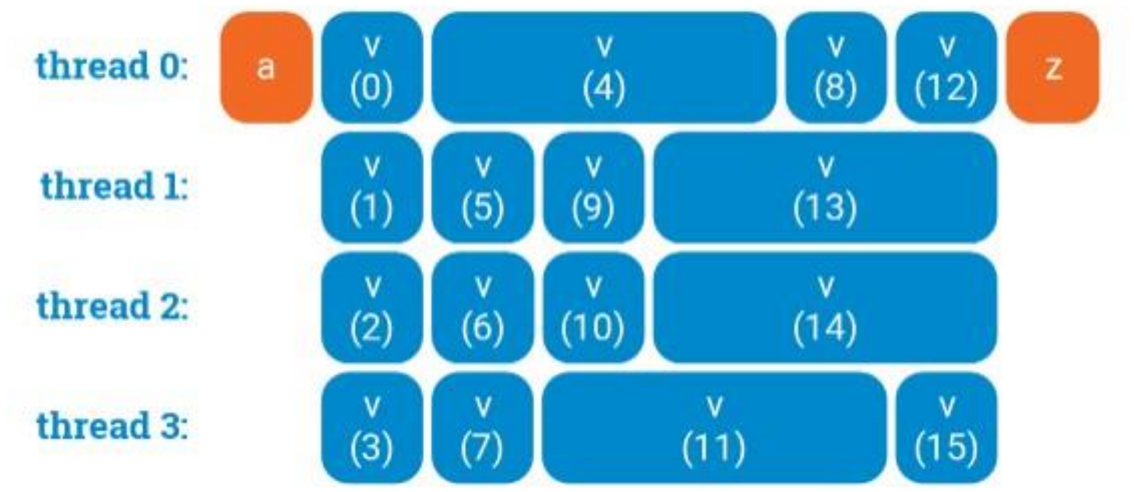
In such a case we can resort to dynamic scheduling. In dynamic scheduling, each thread will take one iteration, process it, and then see what is the next iteration that is currently not being processed by anyone. This way it will never happen that one thread finishes while other threads have still lots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



Reduction Clause:

The usage of the reduction clause is reduction (operator: variable list). The variables in the list are implicitly specified as being private to threads. The operator can be one of +, *, -, &, |, ^, &&, and ||.

Example#2: Parallelizing Array Addition using static scheduling and reduction

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

#define ARRAY_SIZE 10

int main() {
    int array[ARRAY_SIZE];
    int sum = 0;

    // Initialize the array with some values
    for (int i = 0; i < ARRAY_SIZE; i++) {
        array[i] = i + 1;
    }

    // Parallelize the array addition using OpenMP with static scheduling
    #pragma omp parallel for schedule(static) reduction(+:sum)
    for (int i = 0; i < ARRAY_SIZE; i++) {
        sum += array[i];
    }

    // Print the result
    printf("Array: ");
    for (int i = 0; i < ARRAY_SIZE; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    printf("Sum: %d\n", sum);

    return 0;
}
```

Example#3: Parallelizing Vector Addition using Dynamic scheduling and reduction

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

#define ARRAY_SIZE 10

int main() {
    int  array1[ARRAY_SIZE];
    int  array2[ARRAY_SIZE];
    int result[ARRAY_SIZE];

    // Initialize arrays with some values
    for (int i = 0; i < ARRAY_SIZE; i++) {
        array1[i] = i;
        array2[i] = 2 * i;
    }

    // Parallelize the vector addition using OpenMP with dynamic scheduling
    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < ARRAY_SIZE; i++) {
        result[i] = array1[i] + array2[i];
    }
    // Print the result
    printf("Array 1: ");
    for (int i = 0; i < ARRAY_SIZE; i++) {
        printf("%d ", array1[i]);
    }
    printf("\n");

    printf("Array 2: ");
    for (int i = 0; i < ARRAY_SIZE; i++) {
        printf("%d ", array2[i]);
    }
    printf("\n");

    printf("Result: ");
    for (int i = 0; i < ARRAY_SIZE; i++) {
        printf("%d ", result[i]);
    }
    printf("\n");

    return 0;
}
```

***Task#2**: Write a program in openMP C to increase the array size from 1000, 3000, 5000…..10000 in Example#2 with increasing chunk size in static and dynamic scheduling. Calculate the time of both scheduling with chunk size 1 and 2 and compare their performance.*