Practical 3: Comparing OpenMP scheduling

GAHAN SARAIYA (18MCEC10)

18mcec10@nirmauni.ac.in

I. AIM

Write a parallel program using OpenMP to explore different impact of following scheduling.

II. INTRODUCTION

OpenMP specialty is a parallelization of loops. The loop construct enables the parallelization.

```
#pragma omp parallel for
for (...) {
    ...
}
```

OpenMP then takes care about all the details of the parallelization. It creates a team of threads and distributes the iterations between the threads.

To change this behavior OpenMP provides various explicit method for scheduling listed below:

- static
- dynamic
- guided
- auto
- runtime

I. Explicit Scheduling

```
#pragma omp parallel for schedule(scheduling-type)
for (...) {
...
}
```

then OpenMP uses scheduling-type for scheduling the iterations of the for loop.

II. Runtime

If the `scheduling-type` (in the schedule clause of the loop construct) is equal to runtime then OpenMP determines the scheduling by the internal control variable `run-sched-var`. We can set this variable by setting the environment variable `OMP_SCHEDULE` to the desired scheduling type. For example, in bash-like terminals, we can do

```
$ export OMP_SCHEDULE=sheduling-type
```

Another way to specify `run-sched-var` is to set it with `omp_set_schedule` function.

```
...
omp_set_schedule(sheduling-type);
...
```

III. Explicit Scheduling in OpenMP

I. Static

Syntax: `schedule(static, chunk-size)`

- OpenMP divides the iterations into chunks of size chunk-size and it distributes the chunks to threads in a circular order.
- When no `chunk-size` is specified, OpenMP divides iterations into chunks that are approximately equal in size and it distributes at most one chunk to each thread.

Example: parallelized a for loop with 64 iterations and we used 4 threads to parallelize the for loop. Each row of stars in the examples represents a thread. Each column represents an iteration.

I.1

```
schedule(static):
****************
                ****************
                                ****************
                                                ****************
```

This has 16 stars in the first row. This means that the first tread executes iterations $1,2,3,\ldots,16$.
The second row has 16 blanks and then 16 stars. This means that the second thread executes iterations $17,18,19,\ldots,32$. Similar applies to the threads three and four.

I.2

```
schedule(static, 4):
****            ****            ****            ****
    ****            ****            ****            ****
        ****            ****            ****            ****
            ****            ****            ****            ****
```

This divides iterations into 4 chunks of size 16 and it distributes them to 4 threads.

I.3

```
schedule(static, 8):
********                        ********
        ********                        ********
                ********                        ********
                        ********                        ********
```

This divides iterations into 8 chunks of size 16 and it distributes them to 8 threads.

II. Dynamic

Syntax: `schedule(dynamic, chunk-size)`

- OpenMP divides the iterations into chunks of size `chunk-size`. Each thread executes a chunk of iterations and then requests another chunk until there are no more chunks available.
- There is no particular order in which the chunks are distributed to the threads. The order changes each time when we execute the for loop.
- default value for `chunk-size` is 1

II.1

`schedule(dynamic):`

II.2

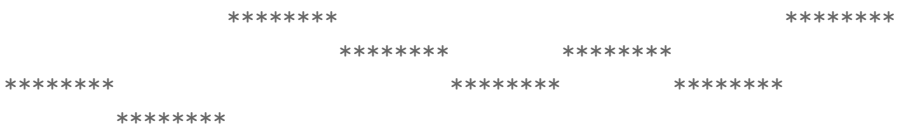
`schedule(dynamic, 1):`

II.3

`schedule(dynamic, 4):`

II.4

`schedule(dynamic, 8):`



III. Guided

Syntax: `schedule(guided, chunk-size)`

guided scheduling type is similar to the dynamic scheduling type

- OpenMP again divides the iterations into chunks. Each thread executes a chunk of iterations and then requests another chunk until there are no more chunks available.

- The difference with the dynamic scheduling type is in the size of chunks. The size of a chunk is proportional to the number of unassigned iterations divided by the number of the threads. Therefore the size of the chunks decreases.
- default value for `chunk-size` is 1

III.1

```
schedule(guided):
                          *********                          *
                 ************                     *******  ***
                                      *******                  *
*****************                             *****       **    *
```

III.2

```
schedule(guided, 2):
                 ************                     ****     **
                                      *******          ***    **
                          *********
*****************                             *****       **    **
```

III.3

```
schedule(guided, 4):
                                      *******
                 ************                     ****    ****
                          *********
*****************                             *****    ****    ***
```

III.4

```
schedule(guided, 8):
                 ************                  ********        ***
*****************
                                      ********
                             *********                ********
```

The size of the chunks is decreasing. First chunk has always 16 iterations. This is because the for loop has 64 iterations and we use 4 threads to parallelize the for loop. If we divide $\frac{64}{4}$, we get 16.

the minimum chunk size is determined in the schedule clause `chunk-size`. The only exception is the last chunk. Its size might be lower then the prescribed minimum size.

The guided scheduling type is appropriate when the iterations are poorly balanced between each other. The initial chunks are larger, because they reduce overhead. The smaller chunks fills the schedule towards the end of the computation and improve load balancing. This scheduling type is especially appropriate when poor load balancing occurs toward the end of the computation.

IV. Auto

Syntax: `schedule(auto)`

The auto scheduling type delegates the decision of the scheduling to the compiler and/or runtime system.

V.

```
schedule(auto):
****************
                ****************
                                ****************
                                                ****************
```

IV. Implementation

Implementing Vector Addition Steps:

- Implement Sequential code
- Implement Parallel code
- Implement function for verification of output (for accuracy calculation)
- Observe time for consumed for execution of sequential and parallel code over 100 iterations (repeating operation for 100 times)
- Calculate speedup for various scheduling type with input size of 10^6 [1]

I. Parallel Code

```
// ------------------------------------------------------------------------------------------
// Author: Gahan Saraiya
// GiT: http://github.com/gahan9/
// StackOverflow: https://stackoverflow.com/users/story/7664524
// Website: http://gahan9.github.io/
// ------------------------------------------------------------------------------------------
// Comparing various OpenMP scheduling method

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <omp.h>
#include <math.h>

#define ARRAY_SIZE 1000000
#define REPEAT     100
#define NUM_THREADS 4


void vector_add(double* vector1, double* vector2, double* result) {
    int i;
```

[1] here runtime method is not actually compared and is out of scope of this experiment

```
        int size_chunks = ARRAY_SIZE/NUM_THREADS;
    #pragma omp parallel num_threads(NUM_THREADS)
        {
        // #pragma omp parallel for num_threads(NUM_THREADS)
        // #pragma omp parallel for
                // #pragma omp for schedule(static)
                // #pragma omp for schedule(static, 100)
                // #pragma omp for schedule(dynamic)
                // #pragma omp for schedule(dynamic, size_chunks)
                // #pragma omp for schedule(auto)
                // #pragma omp for schedule(guided)
                #pragma omp for schedule(guided, size_chunks)
        for (i=0; i < ARRAY_SIZE; i++){
                // for(int i=size_chunks*omp_get_thread_num(); i < size_chunks*(omp_get_thread_num()
            result[i] = vector1[i] + vector2[i];
        }
    }
}


double* generate_array(int no_of_elements) {
    // no_of_elements : Generate array of `no_of_elements` number of elements
        double* array = (double*) malloc(no_of_elements*sizeof(double));
        for(int i=0; i<no_of_elements; i++)
                array[i] = rand()%10000;
        return array;
}

int verify(double* vector1, double* vector2) {
    // Verifying accuracy of computation
        double *adder = (double*) malloc(ARRAY_SIZE*sizeof(double));
        double *verifier = (double*) malloc(ARRAY_SIZE*sizeof(double));
        vector_add(vector1, vector2, adder);

        for(int i=0; i<ARRAY_SIZE; i++) {
                verifier[i] = vector1[i] + vector2[i];
    }
        for(int i=0; i<ARRAY_SIZE; i++){
                if(verifier[i] != adder[i]){
                        return 0;
        }
    }
        return 1;
}


int main() {
        // Generate input vectors and destination vector
```

```
        double *vector1 = generate_array(ARRAY_SIZE);
        double *vector2 = generate_array(ARRAY_SIZE);
        double *result_vector = (double*) malloc(ARRAY_SIZE*sizeof(double));

        // Double check vector_add is correct
        if(!verify(vector1, vector2)) {
                printf("vector_add does not match actual result\n");
                return 0;
        }

        // Test framework that sweeps the number of threads and times each
    // runs for iteration REPEAT
        double start_time, run_time;
    start_time = omp_get_wtime();

    for(int j=0; j<REPEAT; j++){
        vector_add(vector1, vector2, result_vector);
    }
    run_time = omp_get_wtime() - start_time;
          printf("To add %d elements of vectots %d thread(s) took %f seconds\n",
          ARRAY_SIZE, omp_get_max_threads() ,run_time);

}
```

V. Analysis

Below is the result of observation over 100 iterations for input size 10^6 for various methods

Method	**Chunk Size**	**Execution Time (in seconds)**
default	-1	0.875356
auto	-1	0.412167
dynamic	-1	3.638568
dynamic	100	0.342733
dynamic	250000	0.333883
guided	-1	0.351849
guided	100	0.338783
guided	250000	0.332249
static	-1	0.399695
static	100	0.423493

Table 1: result of observation over 100 iterations for input size 10^6 for various methods

- Here chunk size -1 represents that no chunk size specified in command
- 100 is specified as number of fixed chunk
- 250000 is the value derived by $\frac{\text{size of array}}{\text{number of threads}}$

VI. Conclusion

Method	Description
static	When `schedule(static,chunk_size)` is specified, iterations are divided into chunks of size `chunk_size`, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number. When no `chunk_size` is specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread. The size of the chunks is unspecified in this case. A compliant implementation of the static schedule must ensure that the same assignment of logical iteration numbers to threads will be used in two loop regions if the following conditions are satisfied: 1) both loop regions have the same number of loop iterations, 2) both loop region shave the same value of `chunk_size` specified, or both loop regions have `nochunk_sizespecified`, 3) both loop regions bind to the same parallel region,and 4) neither loop is associated with a SIMD construct. A data dependence between the same logical iterations in two such loops is guaranteed to be satisfied allowing safe use of the no wait clause.
dynamic	When `schedule(dynamic,chunk_size)` is specified, the iterations are distributed to threads in the team in chunks. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed. Each chunk contains `chunk_size` iterations, except for the chunk the sequentially last iteration , which may have fewer iterations.When no `chunk_size` is specified, it defaults to 1.
guided	When `schedule(guided,chunk_size)` is specified, the iterations are assigned to threads in the team in chunks. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned For a `chunk_size` of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team,decreasing to 1. For a `chunk_size` with value k(greater than 1), the size of each chunk is determined in the same way, with the restriction that the chunks do not contain fewer thank iterations (except for the chunk that contains the sequentially last iteration, which may have fewer thank iterations).
auto	When schedule(auto) is specified, the decision regarding scheduling is delegated to the compiler and/or runtime system. The programmer gives the implementation the freedom to choose any possible mapping of iterations to threads in the team

Table 2: AllInOne of Scheduling types and description

For the example considered in Program described in Section IV.I and results from Table 1 we can conclude that the task is to simply add two numbers for each index element in both the vectors.

- guided scheduling method tooks minimum execution time
- for the dynamic scheduling the same value are updated as `no-of-chunks-defined`.