PAR Laboratory Assignment
Lab 2: Brief tutorial on OpenMP programming model

Alvaro Moreno Ribot, **PAR 2214**
Albert Escoté Alvarez, **PAR 2205**

OpenMP questionnaire

Day 1: Parallel regions and implicit tasks

1.hello.c

1. **How many times will you see the"Hello world!"message if the program is executed with"./1.hello"?**

We will see two times the message “Hello world” because the “#pragma omp parallel” directive creates two threads by default if we don’t specify it.

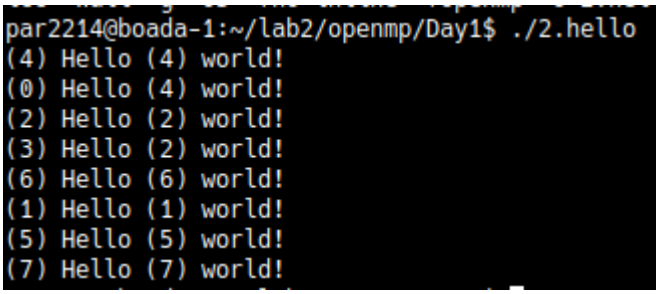
2. **Without changing the program, how to make it print 4 times the "Hello World!" message?**

We need to change the value of the global variable `OMP_NUM_THREADS` and set it to 4. We can use “`OMP_NUM_THREADS=4 ./1.hello`”.

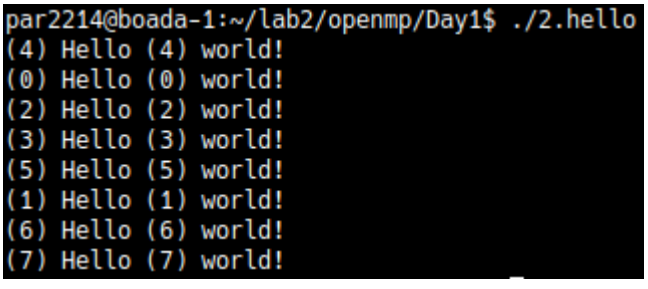
2.hello.c:

1. **Is the execution of the program correct? (i.e., prints a sequence of" (Thid) Hello (Thid) world!" being Thid the thread identifier). If not, add a data sharing clause to make it correct?**

No. As we can see on the screenshot below there are dataraces because we can see that some Thids appear repeated when they shouldn’t be appearing (4, 2).



If we add a ***private(id)*** to the ***#pragma omp parallel*** clausule we can see that the datarace problem is solved. We could also change the declaration of the **id** variable inside the parallel region.



2. **Are the lines always printed in the same order? Why the messages sometimes appear inter-mixed? (Execute several times in order to see this).**

No. The lines are printed in the order threads finish so there will never be a particular order. The reason we sometimes see some inter-mixed lines is because a thread finishes between the two ***printf*** from another thread so it will execute its ***printf*** before the second from the other thread is executed.

3.howmany.c:

Assuming the OMP_NUM_THREADS variable is set to 8 with "OMP_NUM_THREADS=8 ./3.how_many"

1. What does omp_get_num_threads return when invoked outside and inside a parallel region?

As we can see on the screenshot on the right, when we are outside the parallel region, ***omp_get_num_threads*** returns 1 as there is only 1 thread running. However, when we get inside a parallel region we will get the value of the number of threads defined. By default, as we set OMP_NUM_THREADS=8 we will get 8, but the code modifies the number of threads in some occasions so this is why we get a different number in some parallel regions.

```
par2214@boada-1:~/lab2/openmp/Day1$ OMP_NUM_THREADS=8 ./3.how_many
Starting, I'm alone ... (1 thread)
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the first parallel (8)!
Hello world from the szecond parallel (4)!
Hello world from the szecond parallel (4)!
Hello world from the szecond parallel (4)!
Hello world from the szecond parallel (4)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
Hello world from the third parallel (8)!
```

2. Indicate the two alternatives to supersede the number of threads that is specified by the OMP_NUM_THREADS environment variable.

We can supersede the number of threads by adding the clause “num_threads(X)” when we call the “#pragma omp parallel” directive , being “X” the number of threads we want. Or we can also add the “omp_set_num_threads(X)” function inside the code before we call the parallelization.

3. Which is the lifespan for each way of defining the number of threads to be used?

The lifespan for the first way of defining the number of threads (adding the clause) is the region between “{ }” if we add them, if not, the lifespan is only the next code line.
For the second way of defining the number of threads (adding the function), it modifies the OMP_NUM_THREADS variable so it keeps that value until it is superseded again or the program ends.

4.datasharing.c

1. Which is the value of variable x after the execution of each parallel region with different data-sharing attributes (shared, private, first private and reduction)? Is that the value you would expect? (Execute several times if necessary)

shared
We have datarace on the variable x so we can’t guarantee that the variable will always have the same value. In our executions we got 120 and 125 values for the variable x.

private
This time the variable is local for each thread, so we will always see the global (shared) variable x and never the local variable x from the private.

first-private
This time the variable is initialized to the same value as the shared variable. This makes no effect on the result we get as we will still be only seeing the shared variable x which has never been changed.

reduction
This time, the variable gets initialized as if it was private and at the end they get all reduced to a the **sum** variable. This is a correct approach.

5.datarace.c

1. Is the program executing correctly? Why?

No, the program is not executing correctly because the if condition inside the bucle can be executed simultaneously by all the threads. This can make that one thread overwrite the value of "maxvalue" that another thread writes, being the new value of the attribute smaller than the one before.

2. Propose two alternative solutions to make it correct, without changing the structure of the code (just add directives or clauses). Explain why they make the execution correct.

One possible solution is to add the "pragma omp critical" directive to the hole if condition, using "{ }". With this solution we ensure that the if condition is executed separately by each thread. This solves the problem exposed in the first question but it makes the program so slow because it waste a lot of time blocking and unblocking the execution as it's done by software.

Another possible solution is to add the "reduction(max:maxvalue)" clause after the "#pragma omp parallel private(i)" directive. This clause accumulates the local maxvalue for each thread and when the directive ends it compares all the values and gets the bigger one.

3. Write an alternative distribution of iterations to implicit tasks (threads) so that each of them executes only one block of consecutive iterations (i.e. N divided by the number of threads).

```
int main()
{
    int i, maxvalue=0;

    omp_set_num_threads(8);
    #pragma omp parallel private(i) reduction(max:maxvalue)
    {
    int id = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    int N_each_thread = N/howmany;

    for (i=id*N_each_thread; i < (id+1)*N_each_thread; i++) {
        if (vector[i] > maxvalue)
           maxvalue = vector[i];
    }
    }

    if (maxvalue==15)
         printf("Program executed correctly - maxvalue=%d found\n", maxvalue);
    else printf("Sorry, something went wrong - incorrect maxvalue=%d found\n",
maxvalue);

    return 0;
}
```

6.datarace.c

1. Is the program executing correctly? Why?

No, as we can see in the image below, after executing the same program some times it failed. This is due to the fact that all threads can execute the "countmax++;" line at the same time so one thread can overwrite the other.

```
par2205@boada-1:~/lab2/openmp/Day1$ ./6.datarace
Program executed correctly - maxvalue=15 found 3 times
par2205@boada-1:~/lab2/openmp/Day1$ ./6.datarace
Program executed correctly - maxvalue=15 found 3 times
par2205@boada-1:~/lab2/openmp/Day1$ ./6.datarace
Program executed correctly - maxvalue=15 found 3 times
par2205@boada-1:~/lab2/openmp/Day1$ ./6.datarace
Program executed correctly - maxvalue=15 found 3 times
par2205@boada-1:~/lab2/openmp/Day1$ ./6.datarace
Sorry, something went wrong - incorrect maxvalue=15 found 1 times
```

2. Propose two alternative solutions to make it correct, without changing the structure of the program (just using directives or clauses) and never making use of critical. Explain why they make the execution correct.

On the one hand we can use the "#pragma omp atomic" directive the line before we execute "countmax++;". This directive is similar to "critical" but the difference is that "atomic" is executed by hardware making it, usually, more efficient. It solve the problem exposed in the first question because we ensure that every thread executes the "countmax++;" line separately.

On the other hand we can use the "reduction(+:countmax)" clause after the "#pragma omp parallel private(i)" directive. This clause accumulates the local countmax attribute for each thread and when the directive ends it makes the sum of all the values of all the threads.

7.datarace.c

1. Is the program executing correctly? If not, explain why it is not providing the correct result for one or the two variables (countmax and maxvalue)

No, it is not correctly executing. The variable ***countmax*** has a datarace. The variable ***maxvalue*** is correct because of the reduction clause, but when we sum the variable ***countmax*** we are adding the counter value of all the threads, which could have a different max value.

```
par2214@boada-1:~/lab2/openmp/Day1$ ./7.datarace
Sorry, something went wrong - maxvalue=15 found 9 times
```

2. Write a correct way to synchronise the execution of implicit tasks (threads) for this program.
To solve this we could add a global vector which counts the times every max value have been seen for all the threads. When the loop ends we assign to the ***countmax*** variable the value of the vector in the ***maxvalue*** position:

```
int main()
{
        int i, maxvalue=0;
```

```
        int countmax = 0;
        int v_count[N] = {};

        omp_set_num_threads(8);
        #pragma omp parallel private(i) reduction(max: maxvalue)
        {
        int id = omp_get_thread_num();
        int howmany = omp_get_num_threads();

        for (i=id; i < N; i+=howmany) {
        if (vector[i]==maxvalue)
        #pragma omp atomic
        v_count[maxvalue]++;
        if (vector[i] > maxvalue) {
        maxvalue = vector[i];
        #pragma omp atomic
        v_count[maxvalue]++;
        }
        }
        }
        countmax = v_count[maxvalue];

        if ((maxvalue==15) && (countmax==3))
        printf("Program executed correctly - maxvalue=%d found %d times\n", maxvalue, countmax);
        else printf("Sorry, something went wrong - maxvalue=%d found %d times\n", maxvalue, countmax);

        return 0;
}
```

8.barrier.c

1. Can you predict the sequence of printf in this program? Do threads exit from the #pragma omp barrier construct in any specific order?

No. We can't predict it as the tasks are assigned to threads without any particular order. We can only predict the wake-up part as the times correspond with the task number (ascending).

No, they exit as they finish their work. There's a barrier at which they will all start executing at the same time (when sleep is finished for all), but it doesn't mean they will be printed in any particular order.

Day 2: Explicit tasks

1.single.c

1. **What is the nowait clause doing when associated to single?**

The nowait makes other threads continue their execution without waiting for the thread executing the single instruction to end. All threads execute the for loop but each iteration is assigned once and only to one thread available.

2. **Then, can you explain why all threads contribute to the execution of the multiple instances of single? Why those instances appear to be executed in bursts?**

Because ***#pragma omp parallel*** is called right before the loop so all threads available will execute it, but the ***#pragma omp single nowait*** inside the loop causes the instruction inside the for loop to be executed only once, by any available thread.

The 4 threads need to wait 1 second as there is a ***sleep(1)*** call. This explains the bursts as the first 4 iterations are executed each by only one thread all at once (nowait) and then they wait 1 second.

2.fibtasks.c

1. **Why all tasks are created and executed by the same thread? In other words, why the program is not executing in parallel?**

There is no parallel region being created (#pragma omp parallel). This means that there will be only one thread creating tasks and this same thread will be the one working on them.

2. **Modify the code so that tasks are executed in parallel and each iteration of the while loop is executed only once.**

We will create a parallel region using ***#pragma omp parallel*** before the loop. We will also need a ***#pragma omp single*** directive so that the tasks are created by only one thread (but executed in parallel).

We found that the variable called ***p*** (a pointer) is computed by each task so it must be private to avoid data races. Note that the initial value of ***p*** is assigned outside the scope of the loop, so we need to make sure that each private copy of the ***p*** variable is also initialized to the right value, we achieve this using the ***firstprivate(p)*** directive.

3. **What is the firstprivate(p) clause doing? Comment it and execute again. What is happening with the execution? Why?**

The firstprivate(p) clause specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable, because it exists before the parallel construct.

Now, with the firstprivate(p) clause commented, all threads are taking the same “p” variable so there is a data race on it.

3.taskloop.c

1. **Which iterations of the loops are executed by each thread for each task grainsize or num_tasks specified?**

In the first loop, the one with the grainsize clause, the distributing thread divides the number of iterations between the grainsize specified. The distributed iterations are consecutively inside the task. In this example, we have 12 iterations and the value of the grainsize is 4, so the distributing thread will create 3 tasks of 4 iterations.

In the second loop, the one with the num_tasks clause, the distributing thread divides the number of iterations between the number of tasks specified. The distributed iterations are consecutively inside the task. In this example, we have 12 iterations and the value of the number of threads is 4, so the distributing thread will create 4 tasks of 3 iterations.

2. **Change the value for grainsize and num_tasks to 5. How many iterations is now each thread executing? How is the number of iterations decided in each case?**

In the first case, each thread is now executing 6 iterations. This happens because the grainsize clause divides the iterations between the value of grainsize, but the grainsize can be bigger if the rest of the division is different to 0. The grain size can be increased to less than double.

In the second case, each thread is now executing 2 or 3 iterations. This happens because the number of tasks specified is fixed, so there will always be five tasks, unless there are more tasks than iterations. In that case there will be the same number of tasks than iterations. Going back to this example, as we have 12 iterations and 5 tasks we will have 2 tasks of 3 iterations and 3 tasks of 2 iterations.

3. **Can grainsize and num_tasks be used at the same time in the same loop?**

Yes. The loop would be divided into as many tasks specified with the num_task clause and these tasks would have the size of the value specified in the grainsize clause. This would happen if the number of iterations is bigger than the value of "num_tasks" and "grainsize". If not, they would get the value of the number of iterations.

4. **What is happening with the execution of tasks if the nogroup clause is uncommented in the first loop? Why?**

The nogroup clause on the taskloop removes the implicit taskgroup in the taskloop construct, so each thread can go for free taking the first task found.

4.reduction.c

1. **Complete the parallelisation of the program so that the correct value for variable sum is returned in each printf statement. Note: in each part of the 3 parts of the program, all tasks generated should potentially execute in parallel.**

```
int main()
{
    int i;

    for (i=0; i<SIZE; i++){
    X[i] = i;
    }
    omp_set_num_threads(4);
    #pragma omp parallel
    #pragma omp single
    {
        // Part I
    #pragma omp taskgroup task_reduction(+: sum)
        {
        for (i=0; i< SIZE; i++)
        #pragma omp task firstprivate(i) in_reduction(+: sum)
            sum += X[i];
```

```
    }

        printf("Value of sum after reduction in tasks = %d\n", sum);

        // Part II
    #pragma omp taskloop grainsize(BS) firstprivate(sum)
    for (i=0; i< SIZE; i++)
        sum += X[i];

        printf("Value of sum after reduction in taskloop = %d\n", sum);

        // Part III
    #pragma omp taskgroup task_reduction(+: sum)
    {
        #pragma omp taskloop grainsize(BS) firstprivate(sum)
        for (i=0; i< SIZE/2; i++)
            sum += X[i];

        #pragma omp taskloop grainsize(BS) firstprivate(sum)
        for (i=SIZE/2; i< SIZE; i++)
            sum += X[i];
    }
        printf("Value of sum after reduction in combined task and taskloop =
%d\n", sum);
    }

    return 0;
}
```

5.synchtasks.c

1. **Draw the task dependence graph that is specified in this program**

   A, **B** and **C** are independent so they run in parallel. Task **D** depends on task **A** and **B** to start executing so it will start when **A** and **B** are done. Task **E** depends on **D** and **C**, so it will need to wait until both are completed to execute.

```
MAIN
|
|--> A|
|     |---> D
|--> B|     |
|--> C------|---> E
```

2. **Rewrite the program using only taskwait as task synchronisation mechanism (no depend clauses allowed), trying to achieve the same potential parallelism that was obtained when using depend.**

```c
int a, b, c, d;
int main(int argc, char *argv[]) {
    #pragma omp parallel
    #pragma omp single
    {
        printf("Creating task foo1\n");
        #pragma omp task
        foo1();
        printf("Creating task foo2\n");
        #pragma omp task
        foo2();
        printf("Creating task foo4\n");
        #pragma omp taskwait
        #pragma omp task
        foo4();
        printf("Creating task foo3\n");
        #pragma omp task
        foo3();
        printf("Creating task foo5\n");
        #pragma omp taskwait
        #pragma omp task
        foo5();
    }
    return 0;
}
```

3. **Rewrite the program using only taskgroup as task synchronisation mechanism (no depend clauses allowed), again trying to achieve the same potential parallelism that was obtained when using depend.**

```c
int a, b, c, d;
int main(int argc, char *argv[]) {

    #pragma omp parallel
```

```
    #pragma omp single
    {
        #pragma omp taskgroup
        {
            printf("Creating task foo1\n");
            #pragma omp task
            foo1();
            printf("Creating task foo2\n");
            #pragma omp task
            foo2();
        }
        #pragma omp taskgroup
        {
            printf("Creating task foo4\n");
            #pragma omp task
            foo4();
            printf("Creating task foo3\n");
            #pragma omp task
            foo3();
        }
        printf("Creating task foo5\n");
        #pragma omp task
        foo5();
    }
    return 0;
}
```

3.2 Observing Overheads

Synchronization Overheads

We first analyzed how many synchronization operations (critical or atomic) we have in each of the given versions by checking the code.

- *pi_omp_critical.c*, we can see a ***#pragma omp critical*** in the loop, and it will be executed at each iteration.
- *pi_omp_atomic.c*, we can see a ***#pragma omp atomic*** in the loop, and it will be executed at each iteration too.
- *pi_omp_sumlocal.c*, there's only a ***#pragma omp critical*** outside the loop that groups all the sums into a general sum.
- *pi_omp_reduction.c*, there's no critical or atomic region in this file, but as we know reduction uses a method similar to critical to sum up the final result and avoid data races. So with this in mind, we could count these calls as critical so there would be one call for each thread at the end of the loop.

No we executed all these versions with 100.000.000 iterations and using only 1 thread. The main difference between them is that the *pi_omp_critical* version has the biggest overhead, with a huge difference with respect to the other ones:

```
par2214@boada-1:~/lab2/overheads$ cat pi_omp_critical-100000000-1-boada-2.txt
Total overhead when executed with 100000000 iterations on 1 threads: 2689105.0000 microseconds
```

```
par2214@boada-1:~/lab2/overheads$ cat pi_omp_atomic-100000000-1-boada-2.txt
Total overhead when executed with 100000000 iterations on 1 threads: 4406.0000 microseconds
```

```
par2214@boada-1:~/lab2/overheads$ cat pi_omp_sumlocal-100000000-1-boada-2.txt
Total overhead when executed with 100000000 iterations on 1 threads: 29970.0000 microseconds
```

```
par2214@boada-1:~/lab2/overheads$ cat pi_omp_reduction-100000000-1-boada-2.txt
Total overhead when executed with 100000000 iterations on 1 threads: 34246.0000 microseconds
```

The pi_omp_critical version has a ***#pragma omp critical*** in the loop, so it has a lineal overhead to the number of iterations.

The results executing with the same number of iterations and 4 threads are:

```
par2214@boada-1:~/lab2/overheads$ cat pi_omp_critical-100000000-4-boada-2.txt
Total overhead when executed with 100000000 iterations on 4 threads: 37544703.7500 microseconds

par2214@boada-1:~/lab2/overheads$ cat pi_omp_atomic-100000000-4-boada-3.txt
Total overhead when executed with 100000000 iterations on 4 threads: 5059737.7500 microseconds

par2214@boada-1:~/lab2/overheads$ cat pi_omp_sumlocal-100000000-4-boada-3.txt
Total overhead when executed with 100000000 iterations on 4 threads: 10537.5000 microseconds

par2214@boada-1:~/lab2/overheads$ cat pi_omp_reduction-100000000-4-boada-4.txt
Total overhead when executed with 100000000 iterations on 4 threads: 12275.7500 microseconds
```

And the same for 8 threads:

```
par2214@boada-1:~/lab2/overheads$ cat pi_omp_critical-100000000-8-boada-3.txt
Total overhead when executed with 100000000 iterations on 8 threads: 36113366.8750 microseconds

par2214@boada-1:~/lab2/overheads$ cat pi_omp_atomic-100000000-8-boada-2.txt
Total overhead when executed with 100000000 iterations on 8 threads: 5782221.2500 microseconds

par2214@boada-1:~/lab2/overheads$ cat pi_omp_sumlocal-100000000-8-boada-2.txt
Total overhead when executed with 100000000 iterations on 8 threads: 20460.0000 microseconds

par2214@boada-1:~/lab2/overheads$ cat pi_omp_reduction-100000000-8-boada-2.txt
Total overhead when executed with 100000000 iterations on 8 threads: 19891.1250 microseconds
```

Here is a comparative table to visualize all these results and compare them:

Version	1 Thread (µs)	4 Threads (µs)	8 Threads (µs)	S_4	S_8
critical	2689105.0000	37544703.7500	36113366.8750	0.072	0.0744
atomic	4406.0000	5059737.7500	5782221.2500	0.00087	0.000761

sumlocal	29970.0000	10537.5000	20460.0000	2.844	1.46
reduction	34246.0000	12275.7500	19891.1250	2.78	1.72
sequential		-	-	-	-

Critical and atomic are very slow when executed with more than 1 thread, this is explained because only one thread at a time can be computing with critical as the whole instruction is protected. This leads to the execution to behave as if it was sequentially, but with an added overhead. We know that the atomic takes advantage of the hardware (and not software as the critical) which is much quicker as it just protects reads and writes instead of a whole code region (which is what critical does). This results in atomic adding much less overhead than critical for every call.

When compared, the atomic protected part is only when reading and writing to/from the var sum, so all threads can compute in parallel but will have to wait to read and write the result. With atomic we get a slightly better behaviour compared to critical, but we still could achieve better performance with other strategies.

We get better execution times with reduction and sumlocal. This is because the threads only stay waiting after the computation is over and this is when they have to do the reduction of the separate results of each thread. This results in a critical call only executed once per thread (we can assume it for reduction as it does something similar implicitly).

Thread creation and termination

In this section we will be using the *pi_omp_parallel.c* file. Its code computes the time difference between the sequential execution and the parallel execution using a certain number of threads and prints this difference in microseconds, this is the overhead that the parallelization has introduced.

Observing the execution of pi_omp_parallel with 1 iteration and 24 threads, we can observe that we don't have a constant overhead time. Instead it is increasing linearly proportionally to the number of threads. We can assume, from these results, that when you parallelize a program a big overhead is created, but then little overhead is added for each thread created. This explains why when we add threads, the overhead per thread decreases.

```
par2214@boada-1:~/lab2/overheads$ cat pi_omp_parallel-1-24-boada-2.txt
All overheads expressed in microseconds
Nthr    Overhead        Overhead per thread
2       2.0927          1.0463
3       1.5101          0.5034
4       1.6999          0.4250
5       1.9572          0.3914
6       2.1061          0.3510
7       2.0266          0.2895
8       2.2503          0.2813
9       2.2511          0.2501
10      2.4056          0.2406
11      2.4192          0.2199
12      2.6259          0.2188
13      2.9048          0.2234
```

```
14      2.9982          0.2142
15      2.9256          0.1950
16      3.5037          0.2190
17      3.1034          0.1826
18      3.8116          0.2118
19      3.1814          0.1674
20      4.0352          0.2018
21      3.2762          0.1560
22      3.5758          0.1625
23      3.2425          0.1410
24      4.1044          0.1710
```

Task creation and synchronization

In this section we will be using the pi_omp_tasks.c file. This code computes the time difference between the sequential execution and the version that creates the tasks and has the same behaviour as the previous section file.

If we observe the results of this execution we can see that the overhead time does not depend on the task number, it is constant, and is always the same independently of the number of tasks. The time variation could be explained by saying that the machine executing the code could have more workload or anything that could affect this time.

```
par2214@boada-1:~/lab2/overheads$ cat pi_omp_tasks-1-10-boada-2.txt
All overheads expressed in microseconds
Ntasks  Overhead        Overhead per task
2       3.9556          1.9778
4       8.1862          2.0465
6       12.2178         2.0363
8       16.0750         2.0094
10      20.0687         2.0069
12      24.2027         2.0169
14      29.2802         2.0914
16      34.2843         2.1428
18      38.5304         2.1406
20      42.8868         2.1443
22      47.0379         2.1381
24      51.1830         2.1326
26      55.5831         2.1378
28      60.0209         2.1436
30      63.9357         2.1312
32      68.1834         2.1307
34      72.6400         2.1365
36      76.6598         2.1294
38      81.1097         2.1345
40      85.1576         2.1289
42      89.6790         2.1352
44      93.8575         2.1331
46      97.6713         2.1233
48      102.6124        2.1378
50      105.7780        2.1156
52      109.9854        2.1151
54      114.1763        2.1144
56      118.7593        2.1207
```

```
58      124.5195        2.1469
60      134.9962        2.2499
62      138.9281        2.2408
64      143.4402        2.2413
```