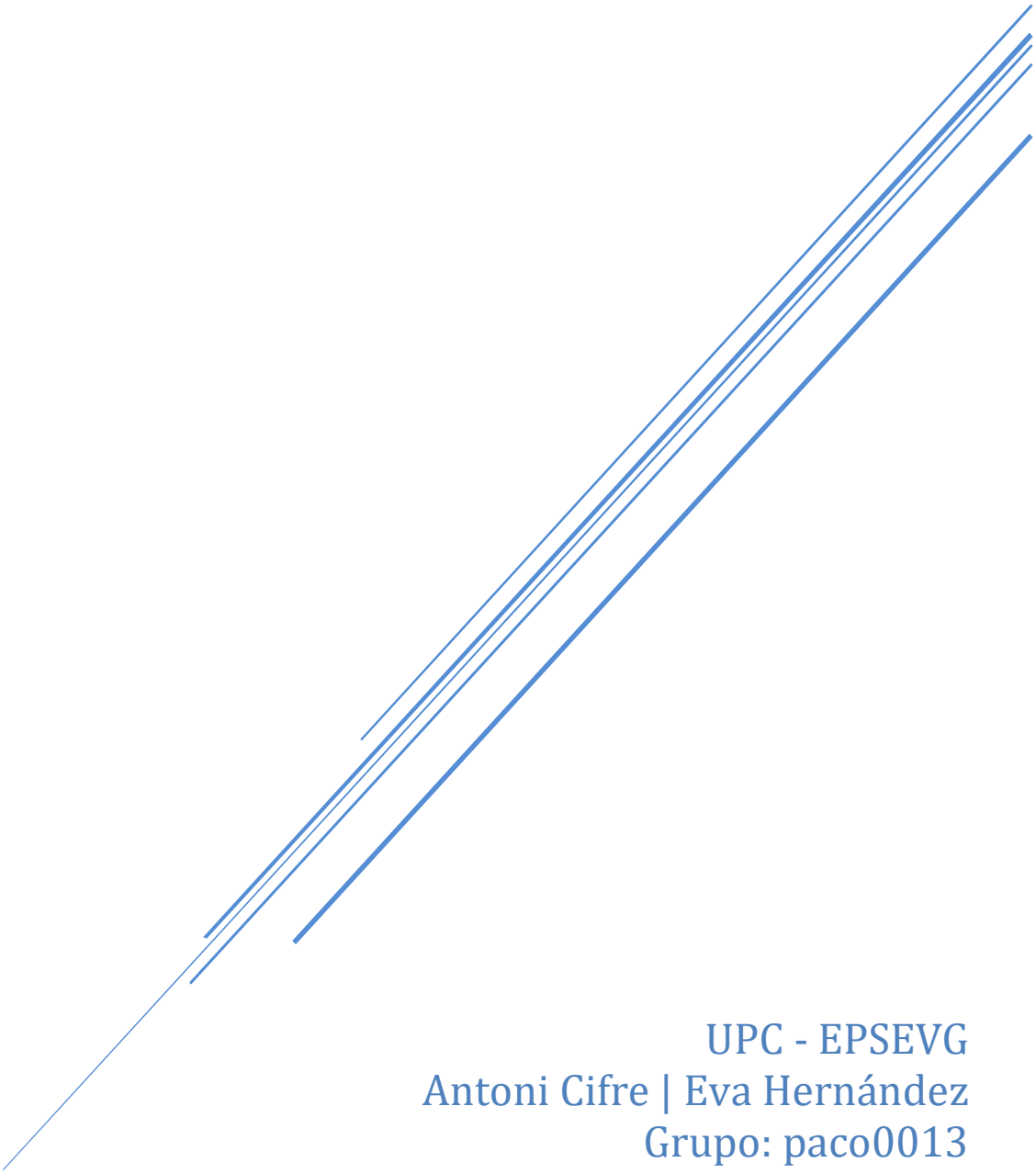
LAB 2

Brief tutorial on OpenMP programming model



UPC - EPSEVG
Antoni Cifre | Eva Hernández
Grupo: paco0013
Fecha: 21/10/2019
3r año, 1r cuatrimestre

Tabla de contenido

3.1 OpenMP questionnaire 2
A) Parallel regions 2
1.hello.c 2
2.hello.c: 2
3.how many.c: 2
4.data sharing.c 3
B) Loop parallelism 3
1.schedule.c 3
2.nowait.c 5
3.collapse.c 6
C) Synchronization 7
1.datarace.c 8
2.barrier.c 8
3.ordered.c 8
D) Tasks 10
1.single.c 10
2.fibtasks.c 10
3.synchtasks.c 11
4.taskloop.c 13
3.2 Observing overheads 15
1.4 Observing overheads 15
2.3 Observing overheads 18
2.3.1 Thread creation and termination 18
2.3.2 Task creation and synchronization 19

3.1 OpenMP questionnaire

When answering to the questions in this questionnaire, please DO NOT simply answer with yes, no or a number; try to minimally justify all your answers and if necessary include any code fragment you need to support your answer. Sometimes you may need to execute several times in order to see the effect of data races in the parallel execution.

A) Parallel regions

1.hello.c

1. How many times will you see the "Hello world!" message if the program is executed with "./1.hello"? 24

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

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

No, porque unos threads se encargan de imprimir la primera línea sin salto y otros se encargan de imprimir la segunda que tiene salto de línea.

3.how many.c:

Assuming the OMP NUM THREADS variable is set to 8 with "export OMP NUM THREADS=8"

1. How many "Hello world ..." lines are printed on the screen?

20

2. What does omp get num threads return when invoked outside and inside a parallel region?

1 thread fuera de la zona paralela(master) y dentro los threads establecidos.

4.data sharing.c

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

shared → da un rango entre 90 y 120 porque dos threads cogen el mismo valor y lo modifican, pero el primer thread lo guarda primero en memoria y el segundo sobreescribe lo que había guardado en memoria el primer thread.

private → 5 Tendrá el valor inicial aleatorio de una posición de memoria porque cuando salga de la zona paralela va a perder el valor calculado.

firstprivate→ 5 Tendrá valor de la variable declarada antes de la zona pero cuando salga de la zona paralela va a perder el valor calculado de dentro.

reduction → 125 Cada thread crea una variable interna donde calculará su parte y luego las pondrá en común con los demás threads y se sumarán todos los resultados de las variables internas de los threads en la variable compartida.

Sí, los resultados eran los esperados.

B) Loop parallelism

1.schedule.c

1. Which iterations of the loops are executed by each thread for each schedule kind?

Going to distribute 12 iterations with schedule(static) ...

Loop 1: (0) gets iteration 0

Loop 1: (0) gets iteration 1

Loop 1: (0) gets iteration 2

Loop 1: (3) gets iteration 9

Loop 1: (3) gets iteration 10

Loop 1: (3) gets iteration 11

Loop 1: (1) gets iteration 3

Loop 1: (1) gets iteration 4

Loop 1: (1) gets iteration 5

Loop 1: (2) gets iteration 6

Loop 1: (2) gets iteration 7

Loop 1: (2) gets iteration 8

En esta distribución hay 12 iteraciones que se reparten entre 4 threads lo que corresponden por cada thread 3 iteraciones..

Going to distribute 12 iterations with schedule(static, 2) ...

Loop 2: (3) gets iteration 6

Loop 2: (3) gets iteration 7

Loop 2: (1) gets iteration 2

Loop 2: (1) gets iteration 3

Loop 2: (1) gets iteration 10

Loop 2: (1) gets iteration 11

Loop 2: (2) gets iteration 4

Loop 2: (2) gets iteration 5

Loop 2: (0) gets iteration 0

Loop 2: (0) gets iteration 1

Loop 2: (0) gets iteration 8

Loop 2: (0) gets iteration 9

En esta distribución se reparten de dos en dos las iteraciones entre los 4 threads y cuando se ha repartido la primera parte, reparte el resto de la iteraciones empezando de nuevo por el thread 0.

Going to distribute 12 iterations with schedule(dynamic, 2) ...

Loop 3: (3) gets iteration 0

Loop 3: (3) gets iteration 1

Loop 3: (3) gets iteration 8

Loop 3: (3) gets iteration 9

Loop 3: (3) gets iteration 10

Loop 3: (3) gets iteration 11

Loop 3: (1) gets iteration 4

Loop 3: (1) gets iteration 5

Loop 3: (2) gets iteration 6

Loop 3: (2) gets iteration 7

Loop 3: (0) gets iteration 2

Loop 3: (0) gets iteration 3

En esta distribución se reparten las iteraciones de dos en dos sin ningún orden de prioridad dentro de los threads y el thread que acaba primero de ejecutar pide los siguientes dos más.

Going to distribute 12 iterations with schedule(guided, 2) ...

Loop 4: (3) gets iteration 0

Loop 4: (2) gets iteration 4

Loop 4: (3) gets iteration 1

Loop 4: (0) gets iteration 2

Loop 4: (0) gets iteration 3

Loop 4: (2) gets iteration 5

Loop 4: (1) gets iteration 6

Loop 4: (1) gets iteration 7

Loop 4: (0) gets iteration 10

Loop 4: (0) gets iteration 11

Loop 4: (3) gets iteration 8

Loop 4: (3) gets iteration 9

En esta distribución se reparten dos por cada uno per cuando algun thread acaba y pide más le puede dar un número de iteraciones superior a lo definido.

2.nowait.c

1. Which could be a possible sequence of printf when executing the program?

Loop 1: thread (0) gets iteration 0

Loop 1: thread (1) gets iteration 1

Loop 2: thread (2) gets iteration 2

Loop 2: thread (3) gets iteration 3

2. How does the sequence of printf change if the nowait clause is removed from the first for directive?

Loop 1: thread (0) gets iteration 0

Loop 1: thread (1) gets iteration 1

Loop 2: thread (0) gets iteration 2

Loop 2: thread (1) gets iteration 3

Cambia porque se espera a que el bucle for acabe y todos los threads acaben.

3. What would happen if dynamic is changed to static in the schedule in both loops? (keepingthe nowait clause)

Loop 1: thread (0) gets iteration 0

Loop 1: thread (1) gets iteration 1

Loop 2: thread (0) gets iteration 2

Loop 2: thread (1) gets iteration 3

Static lo que hará será asignar las tareas a los threads y hasta que estos no acaben de realizar las tareas no les podrá asignar más.

3.collapse.c

1. Which iterations of the loop are executed by each thread when the collapse clause is used?

paco0013@boada-1:~/lab2/openmp/worksharing$ ./3.collapse

(0) Iter (0 0)

(3) Iter (2 0)

(3) Iter (2 1)

(3) Iter (2 2)

(0) Iter (0 1)

(0) Iter (0 2)

(0) Iter (0 3)

(1) Iter (0 4)

(1) Iter (1 0)

(1) Iter (1 1)

(4) Iter (2 3)

(4) Iter (2 4)

(4) Iter (3 0)

(2) Iter (1 2)

(2) Iter (1 3)

(2) Iter (1 4)

(5) Iter (3 1)

(5) Iter (3 2)

(5) Iter (3 3)

(7) Iter (4 2)

(7) Iter (4 3)

(6) Iter (3 4)

(6) Iter (4 0)

(6) Iter (4 1)

(7) Iter (4 4)

El thread 0 hace 4 iteraciones de i y j y los demás threads hacen 3 iteraciones de i y j.

2. Is the execution correct if the collapse clause is removed? Which clause (different than collapse) should be added to make it correct?

Hace repeticiones y se deja iteraciones.

En nuestro caso hemos hecho privadas las variable i y j.

C) Synchronization

1.datarace.c

1. Is the program always executing correctly?

No, el programa no se executa nunca bien.

2. Add two alternative directives to make it correct. Explain why they make the execution correct.

opcion 1:

reduction(+:x)

opcion 2:

```
#pragma omp atomic
    x++;
```

2.barrier.c

1. Can you predict the sequence of messages in this program? Do threads exit from the barrier in any specific order?

Teníamos una idea global de cómo se mostrarán los mensajes, pero no se puede predecir con exactitud porque las posiciones varían entre las diferentes ejecuciones.

En nuestro caso el thread 3 suele salir el primero y los demás en orden aleatoria, pero en algún caso el 3 ha salido en otra posición, por lo cual no podemos afirmar que salgan en un orden determinado.

3.ordered.c

1. Can you explain the order in which the ”Outside” and ”Inside” messages are printed?

No podemos predecir el orden del outside debido a que cada thread empezará en un momento del tiempo no determinado, pero en el caso del inside sabemos que siempre se va a imprimir en orden debido a que se lo especificamos con el pragma omp ordered

2. How can you ensure that a thread always executes two consecutive iterations in order during the execution of the ordered part of the loop body?

Si añadimos un schedule(static) nos aseguramos que reparta las iteraciones del bucle previamente,y como ya sabemos el static suele repartir las iteraciones entre los bucles de uno en uno, asegurándonos que aparecen en orden.

Before ordered - (3) gets iteration 3

Before ordered - (5) gets iteration 5

Before ordered - (4) gets iteration 4

Before ordered - (6) gets iteration 6

Before ordered - (7) gets iteration 7

Before ordered - (1) gets iteration 1

Before ordered - (2) gets iteration 2

Before ordered - (0) gets iteration 0

Inside ordered - (0) gets iteration 0

Inside ordered - (1) gets iteration 1

Inside ordered - (2) gets iteration 2

Before ordered - (0) gets iteration 8

Inside ordered - (3) gets iteration 3

Inside ordered - (4) gets iteration 4

Inside ordered - (5) gets iteration 5

Before ordered - (2) gets iteration 10

Before ordered - (3) gets iteration 11

Before ordered - (4) gets iteration 12

Before ordered - (5) gets iteration 13

Inside ordered - (6) gets iteration 6

Before ordered - (1) gets iteration 9

Before ordered - (6) gets iteration 14

Inside ordered - (7) gets iteration 7

Inside ordered - (0) gets iteration 8

Inside ordered - (1) gets iteration 9

Inside ordered - (2) gets iteration 10

Inside ordered - (3) gets iteration 11

Inside ordered - (4) gets iteration 12

Inside ordered - (5) gets iteration 13

Inside ordered - (6) gets iteration 14

Before ordered - (7) gets iteration 15

Inside ordered - (7) gets iteration 15

D) Tasks

1.single.c

1. Can you explain why all threads contribute to the execution of instances of the single worksharing construct? Why are those instances appear to be executed in bursts?

Todos los threads contribuyen a la ejecución debido al notwait, ya que el thread se queda esperando en el sleep y reparte la siguiente iteración al próximo thread no ocupado.

Se ejecutan a rafagas debido a que los 4 threads se quedan esperando y cuando terminan siguen recorriendo el bucle.

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?

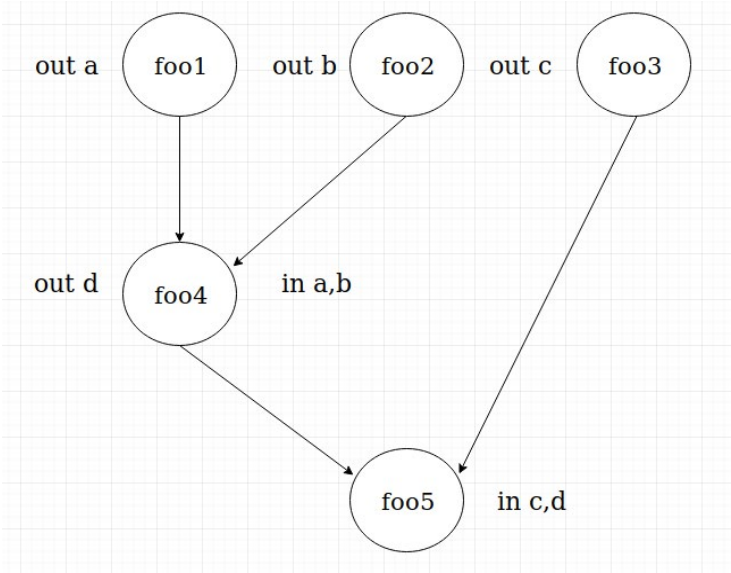
Porque no creamos la zona paralela.

2. Modify the code so that the program correctly executes in parallel, returning the same answer that the sequential execution would return.

```
#pragma omp parallel
{
 #pragma omp single
 {
  while (p != NULL) {
        printf("Thread %d creating task that will compute %d\n",
              omp_get_thread_num(), p->data);
         #pragma omp task firstprivate(p)
           processwork(p);
         p = p->next;
  }
  }
  }
```

3.synchtasks.c

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



2. Rewrite the program using only taskwait as task synchronisation mechanism (no depend clauses allowed)

```
#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 foo3\n");
        #pragma omp task
        foo3();
```

```
            printf("Creating task foo4\n");
            #pragma omp task
            foo4();
    }
    printf("Creating task foo5\n");
    #pragma omp task
    foo5();
}
```

4.taskloop.c

1. Find out how many tasks and how many iterations each task execute when using the grainsize and num tasks clause in a taskloop. You will probably have to execute the program several times in order to have a clear answer to this question.

```
paco0013@boada-1:~/lab2/openmp/tasks$ ./4.taskloop
Going to distribute 12 iterations with grainsize(5) ...
Loop 1: (0) gets iteration 6
Loop 1: (0) gets iteration 7
Loop 1: (0) gets iteration 8
Loop 1: (0) gets iteration 9
Loop 1: (0) gets iteration 10
Loop 1: (0) gets iteration 11
Loop 1: (2) gets iteration 0
Loop 1: (2) gets iteration 1
Loop 1: (2) gets iteration 2
Loop 1: (2) gets iteration 3
Loop 1: (2) gets iteration 4
Loop 1: (2) gets iteration 5
Going to distribute 12 iterations with num_tasks(5) ...
Loop 2: (2) gets iteration 3
```

Loop 2: (2) gets iteration 4

Loop 2: (2) gets iteration 5

Loop 2: (2) gets iteration 6

Loop 2: (1) gets iteration 0

Loop 2: (1) gets iteration 1

Loop 2: (0) gets iteration 10

Loop 2: (3) gets iteration 8

Loop 2: (3) gets iteration 9

Loop 2: (0) gets iteration 11

Loop 2: (1) gets iteration 2

Loop 2: (2) gets iteration 7

2. What does occur if the nogroup clause in the first taskloop is uncommented?

La cláusula nogroup hace la misma función que el notwait, es decir que se ejecutarán los dos taskgroup a la vez.

3.2 Observing overheads

Please explain in this section of your deliverable the main results obtained and your conclusions in terms of overheads for parallel, task and the different synchronisation mechanisms. Include any tables/plots that support your conclusions.

1.4 Observing overheads

1. If executed with only 1 thread and 100.000.000 iterations, do you observe any major overhead in the execution time caused by the use of the different synchronisation mechanisms? You can compare with the baseline execution time of the sequential version in pi sequential.c.

En este primer caso podemos observar que en la ejecución del critical respecto a las otros tiene un overhead bastante elevado, eso es debido a que el critical crea una zona de exclusión la cual solo puede ejecutar un solo thread a la hora y es la causa de este overhead.

2. If executed with 4 and 8 threads and the same number of iterations, do the 4 programs benefit from the use of several processors in the same way? Can you guess the reason for this behaviour?

En este caso, los únicos programas que se ven beneficiados son el reduction y el sumlocal, debido a que el atomic se asegura de que se acceda atómicamente a una zona de memoria para evitar la lectura y escritura simultánea, eso genera que en el caso de la ejecución con un solo thread no se vea afectado el tiempo de ejecución mientras que al añadir más threads dedique el tiempo a bloquear las variables para evitar un error de consistencia, por otra parte, el criticar aumenta considerablemente si tiempo porque cada thread genera una zona crítica y hace que aumente aún más el overhead.

ejecución con 1 thread:

critical:

Total execution time: 4.337176s

Number pi after 100000000 iterations = 3.141592653590426

atomic:

Total execution time: 1.799351s

Number pi after 100000000 iterations = 3.141592653590426

reduction:

Total execution time: 1.832548s

Number pi after 100000000 iterations = 3.141592653590426

su,:

Total execution time: 1.833170s

Number pi after 100000000 iterations = 3.141592653590426

sequential:

Wall clock execution time = 1.793013096 seconds

Value of pi = 3.1415926536

ejecución con 4 threads:

critical:

Total execution time: 36.942143s

Number pi after 100000000 iterations = 3.141592653590201

atomic:

Total execution time: 6.197562s

Number pi after 100000000 iterations = 3.141592653590144

reduction:

Total execution time: 0.472969s

Number pi after 100000000 iterations = 3.141592653589683

sum:

Total execution time: 0.481836s

Number pi after 100000000 iterations = 3.141592653589683

ejecución con 8 threads:

critical:

Total execution time: 36.227995s

Number pi after 100000000 iterations = 3.141592653589771

atomic:

Total execution time: 6.681530s

Number pi after 100000000 iterations = 3.141592653589753

reduction:

Total execution time: 0.259546s

Number pi after 100000000 iterations = 3.141592653589815

sum:

Total execution time: 0.259459s

Number pi after 100000000 iterations = 3.141592653589815

2.3 Observing overheads

2.3.1 Thread creation and termination

3.How does the overhead of creating/terminating threads varies with the number of threads used? Which is the order of magnitude for the overhead of creating/terminating each individual thread in the parallel region?

En este caso podemos decir que el número mínimo ideal de threads para resolver este problema es de 3 threads debido a que es el que tarda menos, y a partir del tercer se genera una carga por división del problema entre los threads la cual en lugar de ayudar en el tiempo de resolución ralentiza la tarea debido a que la tarea se tiene que dividir entre más threads lo cual crea más overheads de división de tareas entre threads pero el tiempo de ejecución no disminuye debido a que se ha llegado al grado óptimo de granulación a partir de tercer thread. En este caso la magnitud va aumentando a medida que el número de threads va aumentando.

All overheads expressed in microseconds

Nthr	Overhead	Overhead per thread
2	1.1799	0.5899
3	0.8624	0.2875
4	1.3455	0.3364
5	1.3611	0.2722
6	1.5028	0.2505
7	1.4994	0.2142
8	1.7015	0.2127
9	1.9736	0.2193
10	1.9907	0.1991
11	2.0506	0.1864
12	2.4888	0.2074
13	2.5867	0.1990
14	2.5864	0.1847
15	2.6202	0.1747

16	2.6505	0.1657
17	3.0266	0.1780
18	2.9979	0.1666
19	3.0110	0.1585
20	3.0207	0.1510
21	3.0356	0.1446
22	3.2142	0.1461
23	3.2131	0.1397
24	12.9668	0.5403

2.3.2 Task creation and synchronization

3. How does the overhead of creating/synchronising tasks varies with the number of tasks created? Which is the order of magnitude for the overhead of creating/synchronising each individual task?

En este caso podemos ver que el overhead aumenta de forma proporcional al número de tareas. La magnitud en este caso es menor que en el caso anterior i que a la larga se estabiliza.

All overheads expressed in microseconds

Ntasks	Overhead	Overhead per task
2	0.1439	0.0720
4	0.5057	0.1264
6	0.7451	0.1242
8	0.9949	0.1244.1
10	1.2370	0.1237
12	1.4783	0.1232
14	1.7194	0.1228
16	1.9495	0.1218
18	2.2106	0.1228
20	2.4438	0.1222
22	2.6885	0.1222

24	2.9348	0.1223
26	3.1775	0.1222
28	3.4336	0.1226
30	3.6600	0.1220
32	3.8908	0.1216
34	4.1470	0.1220
36	4.3905	0.1220
38	4.6351	0.1220
40	4.8745	0.1219
42	5.1199	0.1219
44	5.3703	0.1221
46	5.6072	0.1219
48	5.8276	0.1214
50	6.0838	0.1217
52	6.3449	0.1220
54	6.5728	0.1217
56	6.8169	0.1217
58	7.0600	0.1217
60	7.3111	0.1219
62	7.5279	0.1214
64	7.7719	0.1214