PARAL·LELISME

Laboratori 2: Brief tutorial on OpenMP programming model

PAR1105

Albert Borrellas Maldonado

Víctor Martínez Murillo

23-10-2019

Tardor 2019-2020

A very practical introduction to OpenMP

En aquesta sessió de laboratori fem una introducció a les principals estructures de de OpenMP en extensió al llenguatge C. Aquesta pràctica es divideix en 2 sessions on a la primera observarem diferents versions d'un codi que computa el nombre pi en paral·lel i a la segona ens familiaritzem amb les components principals del model de programació en OpenMP.

OpenMP questionnaire:

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. Escribirà tantes vegades com processadors tingui, en aquest cas, 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 és correcte l'execució del programa ja que es repeteixen valors de *id* a causa de que hi ha *data race*. Per solucionar-ho podríem utilitzar *private(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 sempre estaran escrites en el mateix ordre, dependrà del thread que executi primer el codi. Apareixen intercalats perquè a vegades una tasca pot trigar més que una altra que comença més tard.

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?

Quan s'invoca fora de la regió paral·lela només hi ha un print ja que només hi ha un thread. En canvi, quan s'invoca dins es creen diferents threads on el "get_num_thread" pot retornar diferents número de threads.

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)

Després del primer paral·lel (*shared*) el valor de la x és 120.

Després del segon paral·lel (*private*) el valor de la x és 5.

Després del tercer paral·lel (*firstprivate*) el valor de la x és 5.

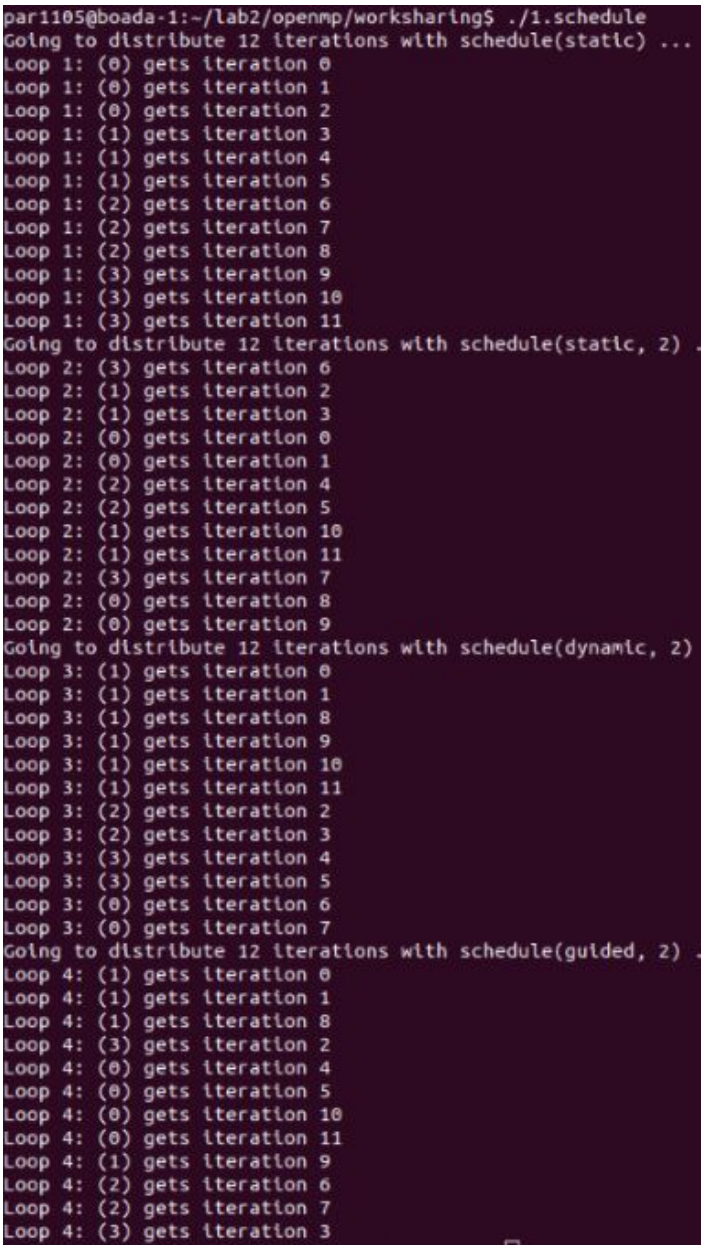
Després del quart paral·lel (*reduction*) el valor de la x és 125.

Ens esperàvem el valor del *private* i *firstprivate* (ja que és el valor de la variable "x" abans d'entrar a la regió paral·lela), però no sabíem quin valor obtindríem amb el *shared* (ja que va variant).

B) Loop parallelism

1.schedule.c

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



Codi 1: Sortida del codi 1.schedule.c

1. Amb el *schedule* static s'assigna en temps de compilació, i com que no se li passa cap argument, per defecte es reparteixen en *chunks* de N/num_threads.

2. Ara els chunks són de mida 2 així que les iteracions es reparteixen en grups de 2 en ordre, quan arribem a la iteració 8 torna a començar.

3. Els threads agafen *chunks* de 2 iteracions dinàmicament fins que s'han acabat totes les iteracions.

4. Els threads tenen un *chunk* major que l'establert a la variable (N = 2) i va decreixent fins a arribar a 2, que és el mínim.

2.nowait.c

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

-Loop 1: thread 0 gets iteration 0 (i=0)

-Loop 2: thread 0 gets iteration 1 (i=3)

-Loop 1: thread 1 gets iteration 1 (i=1)

-Loop 2: thread 1 gets iteration 0 (i=2)

Com s'assigna en temps d'execució, no s'aprofita la utilitat de la clàusula *nowait*.

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

Amb el nowait els for's es poden solapar, quan el trèiem sabem que abans d'executar-se el segon loop, el primer ja haurà acabat. Però segueix sense aprofitar la clàusula nowait ja que estem fent servir un schedule dynamic.

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

Llavors un mateix thread es quedaria amb la mateixa iteració en ambdós loops i així aprofitaríem la sol·lapació.

3.collapse.c

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

```
par1105@boada-1:~/lab2/openmp/worksharing$ ./3.collapse
(0) Iter (0 0)
(0) Iter (0 1)
(0) Iter (0 2)
(0) Iter (0 3)
(0) Iter (0 4)
(3) Iter (3 0)
(3) Iter (3 1)
(3) Iter (3 2)
(3) Iter (3 3)
(3) Iter (3 4)
(1) Iter (1 0)
(1) Iter (1 1)
(1) Iter (1 2)
(4) Iter (4 0)
(2) Iter (2 0)
(2) Iter (2 3)
(2) Iter (2 4)
(1) Iter (1 1)
(4) Iter (4 2)
```

Codi 2: Sortida del codi 3.collapse.c

Les iteracions estàn en grups com si fos un únic loop ja que la clàusula *collapse* distribueix el treball de n loops.

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

No és correcta perquè si trèiem la clàusula *collapse* hi haurà algunes iteracions que es repetiran. La clàusula *ordered* faria correcte el codi canviant-la per *collapse*.

C) Synchronization

1.datarace.c

1. Is the program always executing correctly? No, de fet mai ho fa. Perquè no hi ha sincronització entre els diferents threads.

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

1. Ficant-li la directiva *critical* dins del bucle, on es suma la variable *x* que comparteixen (*shared*) els diferents threads. Ara funciona correctament perquè aquesta directiva obliga a què només un thread pot treballar alhora a la zona *critical*.
2. Ficant ara on està la directiva *critical*, la directiva *atomic*. Té un funcionament similar, però aquesta està enfocada a reservar una localització de memòria (per a fer operacions lectura/escritura) en comptes d'una regió de codi (*critical*).

2.barrier.c

1. Can you predict the sequence of messages in this program? Do threads exit from the barrier in any specific order? No, no es pot. Els missatges canvien l'ordre en diferents execucions. Això és degut a què els threads s'esperen a què tots els altres acabin la feina i arribin a la directiva *barrier*, i a partir d'aquí no hi ha un ordre determinat entre threads i això fa impossible la predicció.

3.ordered.c

1. Can you explain the order in which the "Outside" and "Inside" messages are printed? Entre "outside" i "inside" no hi ha un ordre relatiu ja que s'executa

el bucle paral·lelament amb un *schedule dynamic*, per tant, no es pot predir l'ordre. Però els missatges "inside" estan dins de la directiva *ordered* i això fa que entre ells sí que es respecte l'ordre incremental de les iteracions del bucle.

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?

Canviant el *schedule* a *static* (tenint en compte la directiva *ordered* que ja tenim al codi).

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?

Tots contribueixen perquè hi ha un *nowait* que desfà la barrera implícita del *single* i llavors es van repartint les iteracions. Degut a la línea de codi "sleep(1)".

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?

Perquè només hi ha una crida i és *pragma omp task* que crea tasques però no les paral·lelitza.

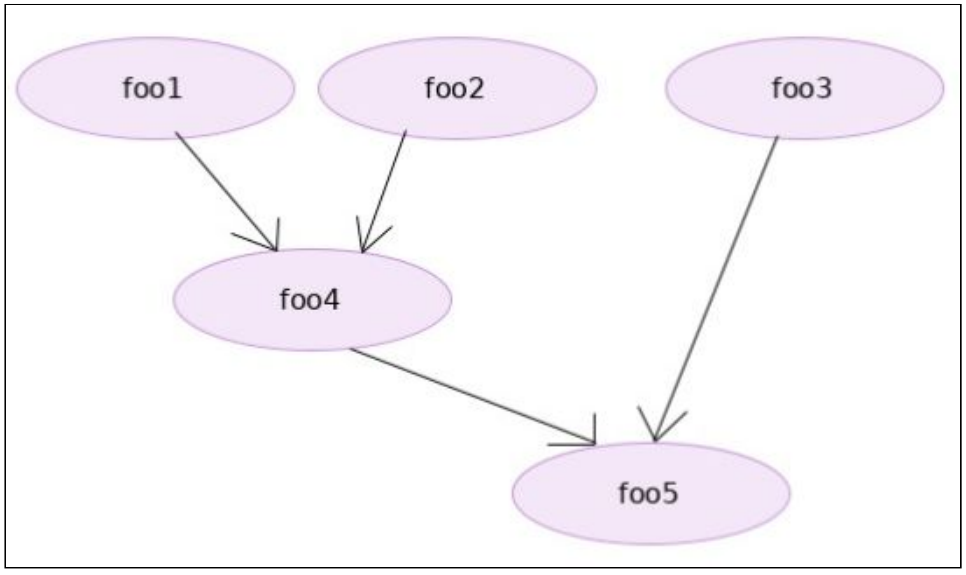
2. Modify the code so that the program correctly executes in parallel, returning the same answer that the sequential execution would return.

```c
struct node *p;

int main(int argc, char *argv[]) {
     struct node *temp, *head;

     printf("Staring computation of Fibonacci for numbers in linked list \n");

     p = init_list(N);
     head = p;

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

               processwork(p);
           p = p->next;
     }
     printf("Finished creation of tasks to compute the Fibonacci for numbers in linked list \n");

     printf("Finished computation of Fibonacci for numbers in linked list \n");
     p = head;
     while (p != NULL) {
        printf("%d: %d computed by thread %d \n", p->data, p->fibdata, p->threadnum);
        temp = p->next;
        free (p);
        p = temp;
     }
     free (p);

     return 0;
}
```

Codi 3: Codi modificat amb les noves directives parallel i single.

3.synchtasks.c

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



Imatge 1: Graf de dependències de tasques del programa 3.synchtasks.c.

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

```
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 foo3\n");
        #pragma omp task
        foo3();
        printf("Creating task foo4\n");
        #pragma omp taskwait
        foo4();
        printf("Creating task foo5\n");
        #pragma omp taskwait
        foo5();
    }
    return 0;
}
```

Codi 4: Codi modificat després de canviar els depends pels taskwaits.

```
par1105@boada-1:~/lab2/openmp/tasks$ ./3.synchtasks
Creating task foo1
Creating task foo2
Creating task foo3
Creating task foo4
Starting function foo1
Starting function foo3
Starting function foo2
Terminating function foo1
Terminating function foo2
Terminating function foo3
Starting function foo4
Terminating function foo4
Creating task foo5
Starting function foo5
Terminating function foo5
```

Imatge 2: Sortida correcte del nou codi, la mateixa que la seqüencial.

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.

Quan utilitzem el *grainsize(5)* s'utilitza un thread i fins que aquest no acaba no s'utilitza un altre. Quan utilitzem el *num_task(5)* si que es pot fer servir més d'un thread alhora i, encara que utilitza el *num_tasks* de 5, com que la N és petita no arriba cada thread a fer 5 tasques.

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

Hi ha un *override* al *taskgroup* i s'executen tant *grainsize* com *num_tasks* alhora.

Observing Overheads

La paral·lelització no és ideal, un dels principals problemes són els *overheads*. Aquests poden provenir de diferents fonts com la creació de threads/tasques o la sincronització entre threads/tasques. Anem a fer un anàlisi dels overheads comparant els programes que calculen el nombre pi mitjançant diferents clàusules i amb la variació del nombre de threads.

El codi 5 és la base per calcula el nombre pi, les diferències entre les diferents versions, com s'ha comentat, és la variació de les directives que trobarem dins dels bucles pel tractament de la sincronització entre threads i/o tasques.

```
double getusec_() {
        struct timeval time;
        gettimeofday(&time, NULL);
        return ((double)time.tv_sec * (double)1e6 + (double)time.tv_usec);
}

#define START_COUNT_TIME stamp = getusec_();
#define STOP_COUNT_TIME(_m) stamp = getusec_() - stamp;\
                        stamp = stamp/1e6;\
                        printf ("%s: %0.6fs\n",(_m), stamp);

int main(int argc, char *argv[]) {
    double stamp;
    double x, sum=0.0, pi=0.0;
    double step;

    const char Usage[] = "Usage: pi_omp_atomic <num_steps> <num_threads>\n";
    if (argc < 3) {
        fprintf(stderr, Usage);
        exit(1);
    }
    long int num_steps = atoi(argv[1]);
    step = 1.0/(double) num_steps;
    int num_threads = atoi(argv[2]);

    START_COUNT_TIME;

    #pragma omp parallel private(x) num_threads(num_threads)
    {
        #pragma omp for
        for (long int i=0; i<num_steps; ++i) {
            x = (i+0.5)*step;
            #pragma omp atomic
            sum += 4.0/(1.0+x*x);
        }
    }
    pi = step * sum;

    STOP_COUNT_TIME("Total execution time");

    /* print results */
    printf("Number pi after %ld iterations = %.15f\n", num_steps, pi);

    return EXIT_SUCCESS;
}
```

Codi 5: Codi que calcula el nombre pi.

Abans de veure les gràfiques de comparació dels temps d'execució de les diferents directives, cal esmentar que la versió seqüencial del codi s'executa aproximadament en 1.7 s.

També un breu resum del que fan aquestes directives en aquest codi:

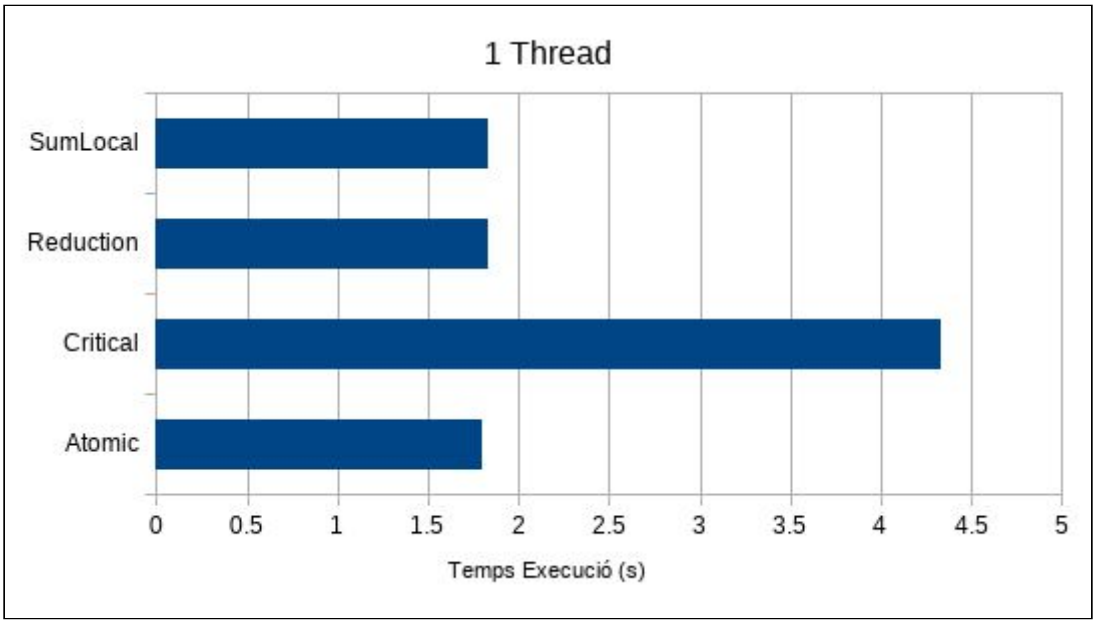
-CRITICAL: Protegeix cada accés a la variable **sum** fent aquest accés exclusiu

-ATOMIC: Garantitza accés indivisible a la localització de memòria on **sum** está guardada.

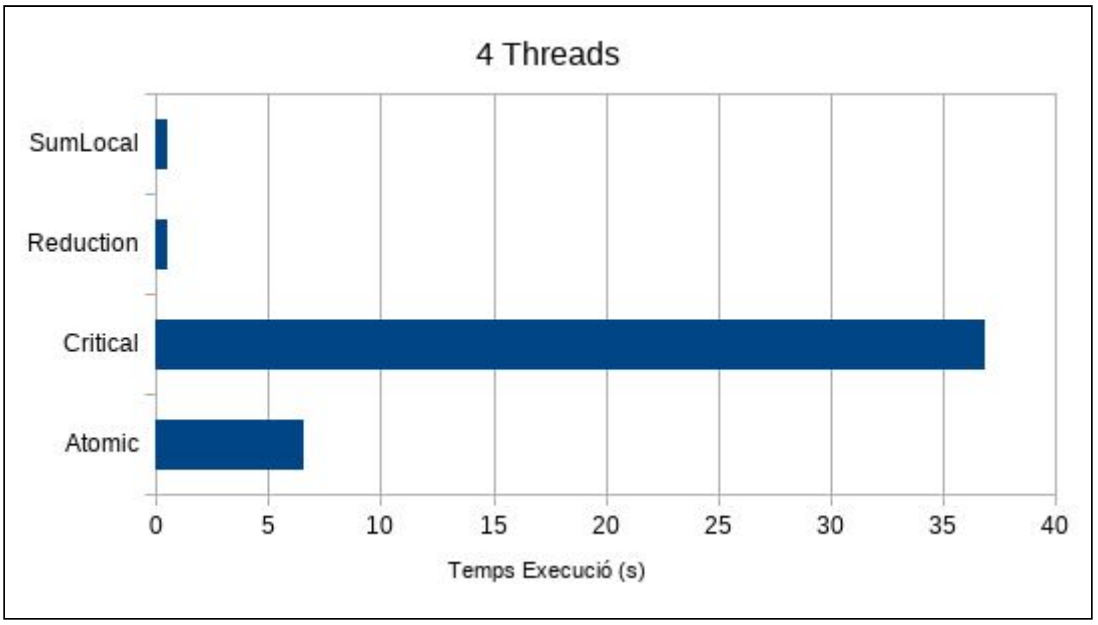
-REDUCTION: Usa les propietats de reduction, és a dir, tots els threads acumulen valors parcials en còpies privades de **sum** i a l'acabar l'execució de la regió afectada, el compilador s'encarrrega d'actualitzar la variable global de forma segura.

-SUMLOCAL: És similar a reduction però l'actualització final es fa a una regió crítica.

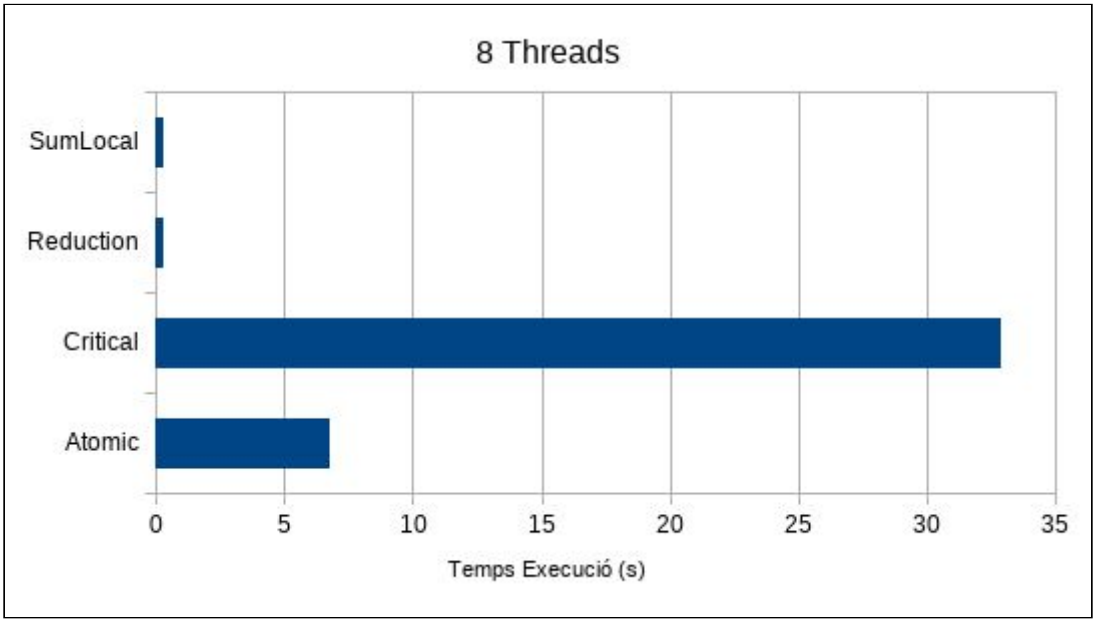
Ara sí, anem a analitzar les gràfiques representades a les imatges 3, 4 i 5:



Imatge 3: Comparació del temps d'execució (s) de les diferents versions del càlcul de pi amb un thread.



Imatge 4: Comparació del temps d'execució (s) de les diferents versions del càlcul de pi amb quatre thread.



Imatge 5: Comparació del temps d'execució (s) de les diferents versions del càlcul de pi amb vuit thread.

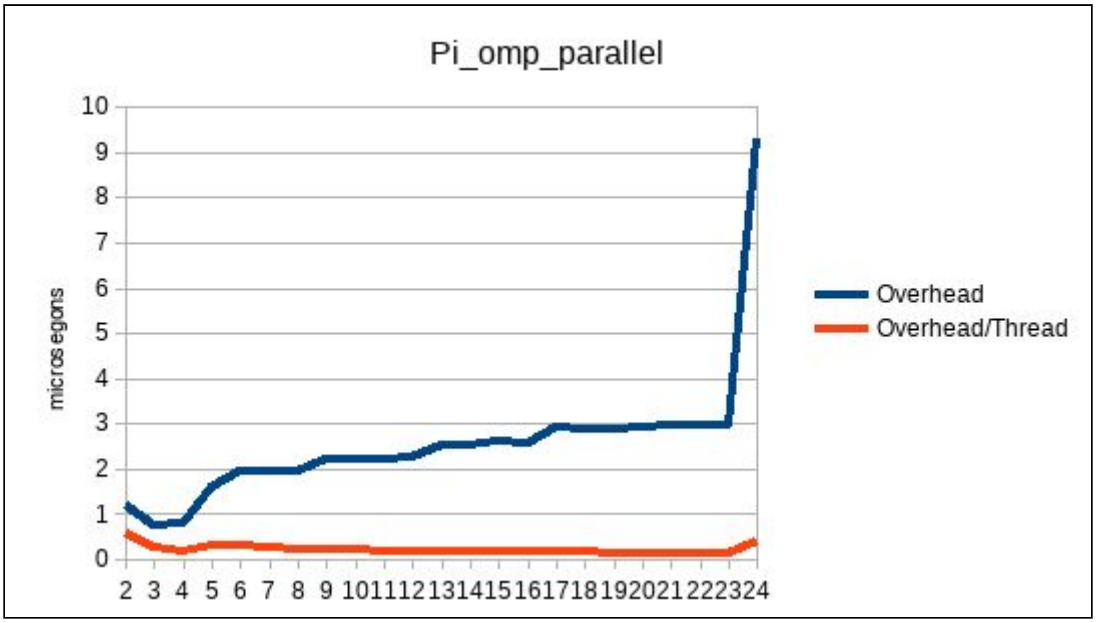
Com podem veure, l'única versió que augmenta el temps d'execució de forma dràstica és la *critical*. Després, també destaca amb 4 i 8 threads, és la versió amb directiva *atomic* la qual té més temps d'execució que *sumlocal* i *reduction.*

La conclusió és que les versions critical i atomic tenen molt *overhead* ja que al treballar en garantitzar l'accés segur a les posicions de memòria corresponents, resultarà en esperes per part dels threads per accedir-hi i després la corresponent sincronització un cop han finalitzat tots els threads. A mesura que augmenta el número de threads això pot anar a pitjor, tot i què, com veiem a la comparació de la imatge 4 i 5, sembla que s'estabilitza.

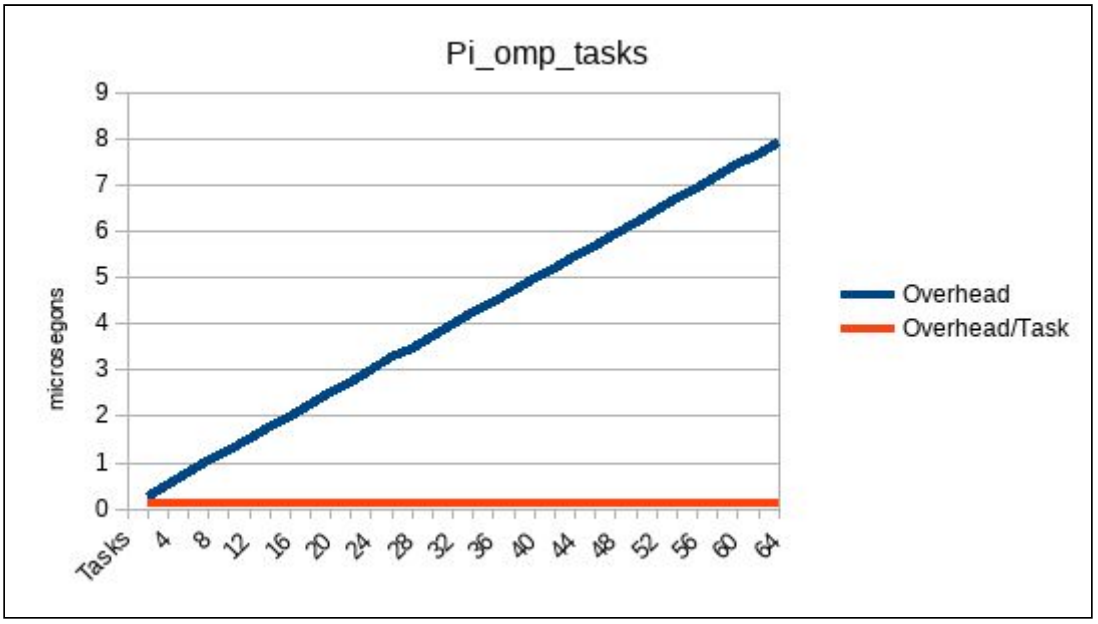
Al codi 6 veiem com calcula el temps d'overhead on s'aplica la directiva. Primer, es calcula el temps del bucle sense aplicar directives d'OpenMP amb un nombre considerable d'iteracions (N = 10.000, en aquest cas) i després es calcula el temps del mateix bucle aplicant-li la directiva corresponent. Amb els dos resultats, es resta el segon temps menys el primer i es divideix entre el nombre d'iteracions per així tenir la mitjana de diferència de temps per iteració i així obtenir un resultat molt acurat.

```c
double getusec_() {
        struct timeval time;
        gettimeofday(&time, NULL);
        return ((double)(time.tv_sec * 1000000L + time.tv_usec));
}

#define NUMITERS 10000

double difference (long int num_steps, int n_threads){
    double x, sum=0.0;
    double step = 1.0/(double) num_steps;

    double stamp1=getusec_();
    for (int iter=0; iter<NUMITERS ; iter++) {
        sum = 0.0;
        for (long int i=0; i<num_steps; ++i) {
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
            }
        }
    stamp1=getusec_()-stamp1;

    omp_set_num_threads(n_threads);
    double stamp2=getusec_();
    for (int iter=0; iter<NUMITERS ; iter++) {
        sum = 0.0;
        #pragma omp parallel private(x) firstprivate(sum)
        for (long int i=0; i<num_steps; ++i) {
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
            }
        }
    stamp2=getusec_()-stamp2;
    return((stamp2-stamp1)/NUMITERS);
}

int main(int argc, char *argv[]) {
    const char Usage[] = "Usage: pi_omp_parallel <num_steps> <max_threads>\n";
    if (argc < 3) {
        fprintf(stderr, Usage);
        exit(1);
    }
    long int num_steps = atoi(argv[1]);
    int max_threads = atoi(argv[2]);

    printf("All overheads expressed in microseconds\n");
    printf("Nthr\tOverhead\tOverhead per thread\n");

    for (int n_threads=2; n_threads<=max_threads; n_threads++) {
        double tmp = difference(num_steps, n_threads);
        printf("%d\t%.4f\t\t%.4f\n", n_threads, tmp, tmp/n_threads);
        }

    return EXIT_SUCCESS;
```

Codi 6 : Codi pi_omp_parallel.c.



Imatge 6: Comparació del temps d'execució (s) de les diferents versions del càlcul de pi amb vuit threads.



Imatge 7: Comparació del temps d'execució (s) de les diferents versions del càlcul de pi amb vuit threads.

Finalment, a les imatges 6 i 7, veiem que hi ha una gran diferencia entre pi_omp executat mitjançant threads que executat amb la creació de tasques. En tasques podem veure que l'overhead és linealment proporcional al nombre de tasques creades. En canvi, amb el que executem amb threads, veiem que el temps d'overhead va variant de forma poc significativa (tenint en compte que l'overhead està mesurat en microsegons) fins al thread 24 (coincidint amb el nombre de threads del boada-2..4) on augmenta dràsticament. L'avantatge de la implementació amb tasques és el bon balanceig de l'overhead entre les tasques (overhead/tasca) on es manté constant, tot i així, la diferència amb la implementació en threads no és gaire significativa parlant de balanceig.

Un cop més hem vist que el paral·lelisme ni és gratuït ni és ideal, per tant, s'ha de ser molt curós per programar aplicant estructures com les del OpenMP.