Paral·lelisme
Laboratori I

Aleix Pérez Vidal i Marc Canals Riba
(par4315)

17 d'octubre de 2019

Regions paral·leles

Hello (I)

```
#pragma omp parallel
printf("Hello world!\n");
```

Qüestió 1

El missatge apareixerà una vegada a cada *thread*. En aquest cas, apareix 24 vegades, el nombre màxim de *threads*.

Qüestió 2

Podem indicar el nombre de *threads*, sense modificar el programa, mitjançant la variable d'entorn `OMP_NUM_THREADS`.

```
OMP_NUM_THREADS=4 ./1.hello
```

Hello (II)

```
#pragma omp parallel num_threads(8)
{
    id = omp_get_thread_num();
    printf("(%d) Hello ", id);
    printf("(%d) world!\n", id);
}
```

Qüestió 1

El missatge no mostra necessàriament el número del *thread* que l'ha imprès ni el mateix número de *thread* en cada part, ja que la variable `id` és compartida i entre l'escriptura i la lectura o entre les dues lectures un altre *thread* la pot haver modificada. Hi podem afegir la *clause* `private`.

```
#pragma omp parallel private(id) num_threads(8)
...
```

Qüestió 2

Les parts del missatge no apareixen necessàriament de manera consecutiva, ja que les crides no constitueixen cap unitat de treball indivisible. Hi podem afegir un *construct* `critical` i ajuntar les crides.

```
#pragma omp critical
{
   printf("(%d) Hello ", id);
   printf("(%d) world!\n", id);
}
```

Hello (III)

```
printf(..., omp_get_num_threads());

#pragma omp parallel
printf(..., omp_get_num_threads());

for (int i = 2; i < 4; i++) {
    omp_set_num_threads(i);
#pragma omp parallel
    printf(..., omp_get_num_threads());
}

printf(..., omp_get_num_threads());

#pragma omp parallel num_threads(4)
printf(..., omp_get_num_threads());

#pragma omp parallel
printf(..., omp_get_num_threads());

printf(..., omp_get_num_threads());
```

Qüestió 1

Si el nombre de *threads* predefinit és 8, el nombre de vegades que apareix cada missatge, és a dir, el nombre de *threads* és el que es mostra a la taula 1.

Nombre de *threads*	**Tipus de regió**	**Definició del nombre de *threads***
1	Seqüencial	
8	Paral·lela	Variable `OMP_NUM_THREADS`
2	Paral·lela	Funció `omp_set_num_threads`
3	Paral·lela	Funció `omp_set_num_threads`
1	Seqüencial	
4	Paral·lela	*Clause* `num_threads`
3	Paral·lela	Funció `omp_set_num_threads`
1	Seqüencial	

Taula 1: Nombre de threads en cada regió del codi.

Qüestió 2

La funció `omp_get_num_threads` retorna el nombre de *threads* que executen la regió en què es crida, tal com podem observar a la qüestio anterior.

Data Sharing

```
omp_set_num_threads(16);

int x = 0;

#pragma omp parallel shared(x)
{
    x += omp_get_thread_num();
}

x = 5;

#pragma omp parallel private(x)
{
    x += omp_get_thread_num();
}

#pragma omp parallel firstprivate(x)
{
    x += omp_get_thread_num();
}

#pragma omp parallel reduction(+:x)
{
    x += omp_get_thread_num();
}
```

Qüestió 1

Després de la regió en què la variable és compartida (`shared`) no podem determinar el valor d'aquesta, ja que depèn de l'ordre en què s'executen les operacions (*data race*).

Després de les regions en què la variable és privada el valor d'aquesta és 5, ja que s'ha modificat el valor de la variable local, inicialitzat a zero a la primera regió (`private`) i a 5 a la segona (`firstprivate`).

Després de la darrera regió (`reduction`), el valor de la variable és $5 + \sum_{i=0}^{15} i$, ja que la operació és realitza correctament.

Paral·lelisme en bucles

Schedule

```
omp_set_num_threads(4);

#pragma omp parallel for schedule(...)
for (i = 0; i < N; i++) ...
```

Qüestió 1

Schedule	***Thread* corresponent a cada iteració**
`static`	0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3
`static, 2`	0, 0, 1, 1, 2, 2, 3, 3, 0, 0, 1, 1
`dynamic, 2`	0, 0, 3, 3, 2, 2, 1, 1, 3, 3, 3, 3
`guided, 2`	3, 3, 2, 2, 0, 0, 1, 1, 3, 3, 3, 3

Taula 2: *Thread* que executa cada iteració en cada *schedule*.

Amb l'*schedule* `guided, 2` la mida del *chunk* disminueix quan el nombre d'iteracions assignades a un *thread* augmenta, sent el minim l'indicat.

Nowait

```
#define N 4
omp_set_num_threads(N);

#pragma omp for schedule(dynamic, 1) nowait
for (i = 0; i < N / 2; i++) ...

#pragma omp for schedule(dynamic, 1) nowait
for (i = N / 2; i < N; i++) ...
```

Qüestió 1

Podem obtenir qualsevol de les possibilitats (des de 0000 fins a 3333), ja que, com que les tasques dels bucles són independents, no cal esperar que acabin les del primer bucle abans de començar les del segon bucle i les podem assignar a qualsevol *thread* disponible, malgrat que és molt poc probable que els *threads* assignats a una tasca del primer bucle siguin disponibles, ja que les tasques són relativament llargues (`sleep`). En una mostra de 100 execucions, observem que hi ha dues possibilitats que destaquen (taula 3).

Ocurrències	***Thread* corresponent a cada iteració**
60	0, 1, 2, 3
30	0, 1, 3, 2
4	0, 2, 1, 3
2	0, 3, 1, 2
3	1, 0, 2, 3
1	1, 0, 3, 2

Taula 3: *Thread* que executa cada iteració.

Qüestió 2

Si eliminem la *clause* `nowait` del primer bucle, les tasques continuen sent relativament llargues (`sleep`), però cal esperar que acabin les del primer bucle abans de començar les del segon bucle, per tant, és molt més probable que els *threads* assignats a una tasca del primer bucle siguin disponibles. En una mostra de 100 execucions, observem que hi ha dues possibilitats que destaquen (taula 4).

Ocurrències	***Thread* corresponent a cada iteració**
51	0, 1, 0, 1
37	0, 1, 1, 0
5	0, 2, 2, 0
4	0, 3, 3, 0
1	1, 0, 0, 1
2	1, 0, 1, 0

Taula 4: *Thread* que executa cada iteració.

Qüestió 3

Si canviem l'*schedule* per `static, 1`, mantenint la *clause* `nowait`, els *threads* sempre són 0, 1, 0, 1, ja que l'assignació d'iteracions al segon bucle és estàtica i espera que els *threads* siguin disponibles.

Collapse

```
#define N 5
omp_set_num_threads(8);

#pragma omp parallel for collapse(2)
for (i=0; i < N; i++)
   for (j=0; j < N; j++) ...
```

Qüestió 1

Amb la *clause* `collapse` es reparteixen les $N_1 * ... * N_M$ iteracions equitativament entre els threads, en ordre creixent de *thread* i d'iteració.

Thread	i	j		***Thread***	i	j
0	0	0		4	2	3
0	0	1		4	2	4
0	0	2		4	3	0
0	0	3		5	3	1
1	0	4		5	3	2
1	1	0		5	3	3
1	1	1		6	3	4
2	1	2		6	4	0
2	1	3		6	4	1
2	1	4		7	4	2
3	2	0		7	4	3
3	2	1		7	4	4
3	2	2				

Taula 5: *Thread* que executa cada iteració.

Qüestió 2

Si eliminem la *clause* `collapse` l'execució no és correcta: per una banda, introduïm interferències entre els *threads*; per altra banda, l'assignació d'iteracions a *threads* no és la mateixa.

Sincronització

Data Race

```
#define N 1 << 20
omp_set_num_threads(8);

#pragma omp parallel for schedule(dynamic,1) private(i) shared(x)
for (i=0; i < N; i++) {
    x++;
}
```

Qüestió 1

És molt poc probable que el resultat del programa sigui l'esperat, ja que la variable `x` és compartida i hi ha interferències entre els *threads*.

Qüestió 2

Per corregir el programa, podem utilitzar el *construct* `critical`, que estableix una regió d'exclusió mútua en què no hi pot treballar més d'un *thread* alhora, o el *construct* `atomic`, que garanteix que una ubicació s'accedeix de manera atòmica. Generalment, la segona és més eficient. D'aquesta manera, la variable `x` s'actualitzaria correctament.

```
...
#pragma omp critical
x++;
...


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

Barrier

```
int id;

#pragma omp parallel private(id) num_threads(4)
{
    int sleeptime;

    id = omp_get_thread_num();
    sleeptime = (2 + id * 3) * 1000;

    printf(...);
    usleep(sleeptime);
    printf(...);
#pragma omp barrier
    printf(...);
}
```

Qüestió 1

No podem predir l'ordre en què es mostren els missatges, ja que coneixem aproximadament el temps d'execució, però no quan comença ni quan acaba. Tot i això, si que podem determinar que cap *thread* traspassarà la barrera fins que no hi siguin tots els altres *threads*. A més, com que l'espera és realtivament llarga, és molt poc probable que un *thread* acabi l'espera abans que un altre *thread* la comenci.

Per altra banda, els *threads* no surten de la barrera en cap ordre específic.

Ordered

```
#define N 16
omp_set_num_threads(8);

#pragma omp parallel for schedule(dynamic) ordered
for (i=0; i < N; i++) {
    printf(...);
#pragma omp ordered
    printf(...);
}
```

Qüestió 1

Els missatges fora la regió ordenada apareixen desordenats, ja que l'execució és en paral·lel; en canvi, els missatges dins la regió ordenada apareixen ordenats temporalment, és a dir, per iteració i de forma creixent.

Qüestió 2

Podem garantir que un *thread* executa dues iteracions consecutives en ordre afegint la *clause* `schedule(dynamic, 2)`.

Tasques

Single

```
#pragma omp parallel num_threads(4)
for (int i = 0; i < 20; i++)
#pragma omp single nowait
{
    printf(...)
    sleep(1);
}
```

Qüestió 1

Tots els *threads* executen el mateix bucle en paral·lel, però, com que s'utilitza el *construct* single, només un dels *threads* executa la regió, per tant, el missatge només apareix 20 vegades.

Per altra banda, sembla que els missatges apareguin en ràfegues de quatre missatges perquè el temps d'execució de la regió és relativament gran (`sleep`) i cada quatre iteracions hem d'esperar que algun *thread* sigui disponible.

Fibonacci Tasks

Qüestió 1

```
while (p != NULL) {
   ...
#pragma omp task
       processwork(p);
   p = p->next;
}
```

El programa no s'executa en paral·lel ja que no s'ha definit cap regió paral·lela. L'únic *thread* existent crea les tasques i les realitza.

Qüestió 2

```
#pragma omp parallel firstprivate(p)
#pragma omp single
while (p != NULL) {
   ...
#pragma omp task
       processwork(p);
   p = p->next;
}
```

El programa s'executa en paral·lel ja que s'ha definit una regió paral·lela. En aquest cas, un *thread* (`single`) crea les tasques i els altres les realitzen.

Synchronized Tasks

```
#pragma omp parallel
#pragma omp single
{
#pragma omp task depend(out:a)
   foo1();
#pragma omp task depend(out:b)
   foo2();
#pragma omp task depend(out:c)
   foo3();
#pragma omp task depend(in: a, b) depend(out:d)
   foo4();
#pragma omp task depend(in: c, d)
   foo5();
}
```

Qüestió 1

A continuació (figura 1) podem veure el graf de dependències de les tasques.

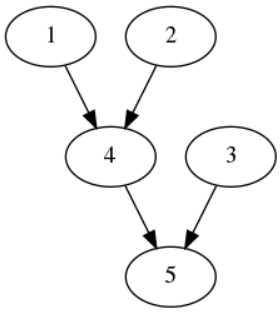

Figura 1: Graf de dependències.

Qüestió 2

Podem reemplaçar les dependències per esperes.

```
#pragma omp parallel
#pragma omp single
{
#pragma omp task
    foo1();
#pragma omp task
    foo2();
#pragma omp taskwait
#pragma omp task
    foo4();
#pragma omp task
    foo3();
#pragma omp taskwait
#pragma omp task
    foo5();
}
```

Taskloop

```
#pragma omp parallel num_threads(4)
#pragma omp single
{
#pragma omp taskloop grainsize(5)
   for (i = 0; i < 12; i++) ...

#pragma omp taskloop num_tasks(5)
   for (i = 0; i < 12; i++) ...
}
```

Qüestió 1

Al primer bucle, les tasques haurien de ser de 5 iteracions, per tant, hi hauria d'haver $12/5 = 2,4$ tasques. En totes les observacions realitzades, utilitzant la funció `usleep`, per tal de reduir la probabilitat que un *thread* executi més d'una tasca i d'aquesta manera poder distingir les tasques pel *thread* que les executa, hi ha 2 tasques de 6 iteracions.

Al segon bucle, hi hauria d'haver 5 tasques, per tant, les tasques haurien de ser de $12/5 = 2,4$ iteracions. En totes les observacions realitzades, utilitzant 5 *threads* i la funció `usleep`, per tal de reduir la probabilitat que un *thread* executi més d'una tasca i d'aquesta manera poder distingir les tasques pel *thread* que les executa, hi ha 3 tasques de 2 iteracions i 2 tasques de 3 iteracions.

Qüestió 2

```
#pragma omp taskloop grainsize(5) nogroup
...
```

Si afegim la *clause* `nogroup` al *construct* del primer bucle, s'ignora la *clause* `taskgroup` implícita, que garanteix que totes les tasques del primer bucle han acabat abans de començar les del segon; ara les tasques del primer bucle i les del segon s'executen en paral·lel.

Overheads

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

pi = step * sum;
```

Versió amb el *construct* `critical`.

```
#pragma omp parallel private(x)
{
#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;
```

Versió amb el *construct* `atomic`.

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

pi = step * sum;
```

Versió amb la *clause* `reduction`.

```
#pragma omp parallel private(x) firstprivate(sumlocal)
{
#pragma omp for
    for (long int i = 0; i < num_steps; i++) {
      x = (i + 0.5) * step;
      sumlocal += 4.0 / (1.0 + x * x);
   }
#pragma omp critical
   sum += sumlocal;
}
```

Versió amb una variable local i el *construct* `critical`.

Observem (figura 2) que amb un únic processador l'*overhead* de totes les versions, excepte la que utilitza el *construct* `critical`, és practicament nul. Per altra banda, les versions que utilitzen els *construct* `critical` i `atomic` són més lentes que la versió seqüencial, ja que, a les regions crítica i atòmica, a més de l'actualització de la variable compartida, s'hi realitzen operacions que es poden fer fora d'aquestes regions; en canvi, les versions restants són més ràpides que la versió seqüencial.

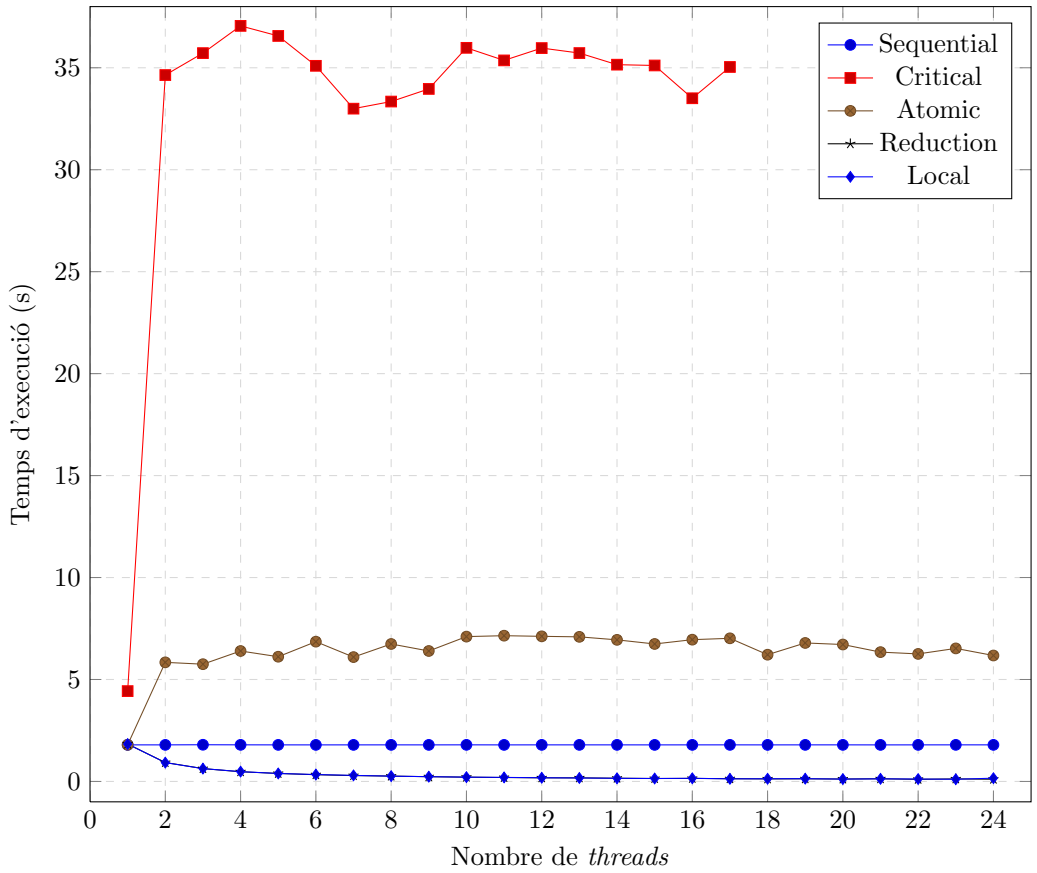

Figura 2: Temps d'execució de cada versió en funció del nombre de *threads*.

L'*overhead* per *thread* tendeix al cost de creació i destrucció d'un *thread* O_{thread}, de l'ordre de les dècimes de microsegon, disminuint quan augmenta el nombre de *threads* P, ja que el cost global de creació i destrucció de *threads* O_{global}, independent del nombre de *threads*, es reparteix entre aquests, cada vegada més nombrosos. Per tant, l'*overhead* per *thread* s'aproxima a la funció $O_{global}/P+O_{thread}$.

Per altra banda, l'*overhead* total és la suma del cost global i el cost de cada *thread*, per tant, s'aproxima a la funció $O_{global} + P * O_{thread}$.

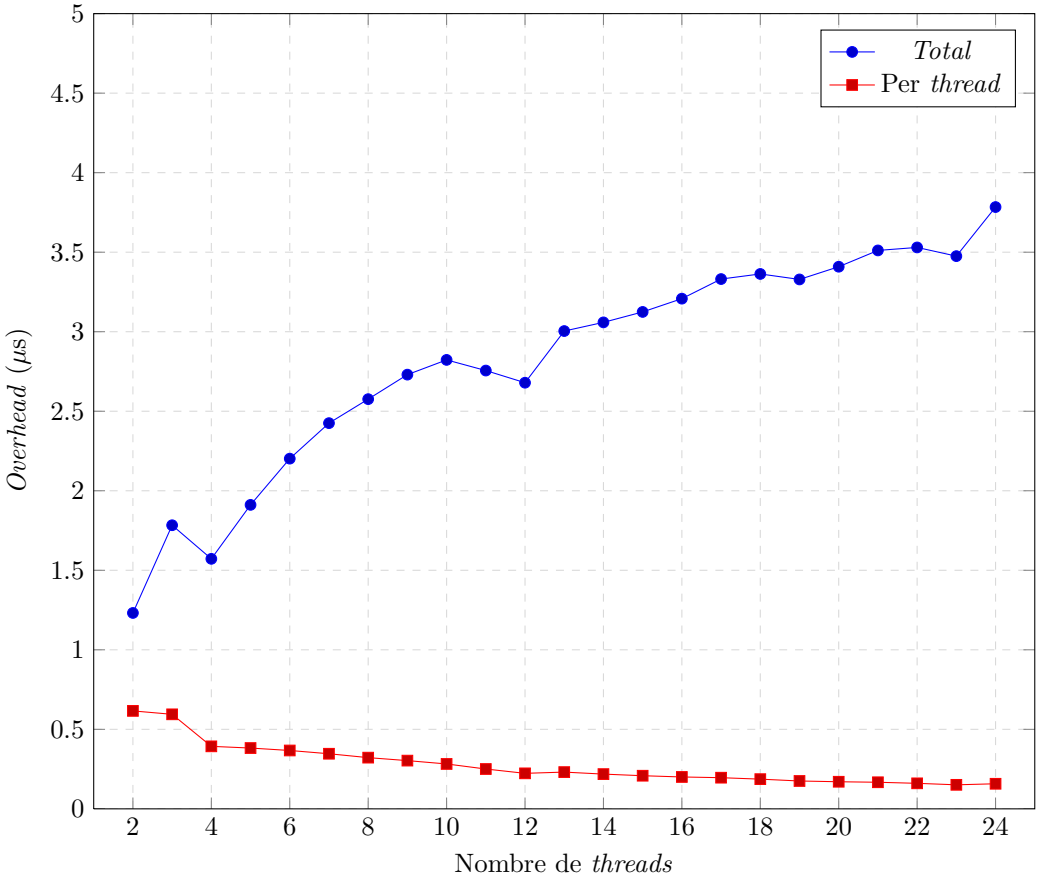

Figura 3: *Overhead* total i per *thread* en funció del nombre de *threads*.

Respecte a l'*overhead* per tasca, observem que és pràcticament constant, també de l'ordre de les dècimes de microsegon. L'*overhead*total és lineal i el terme independent és gairebé nul.

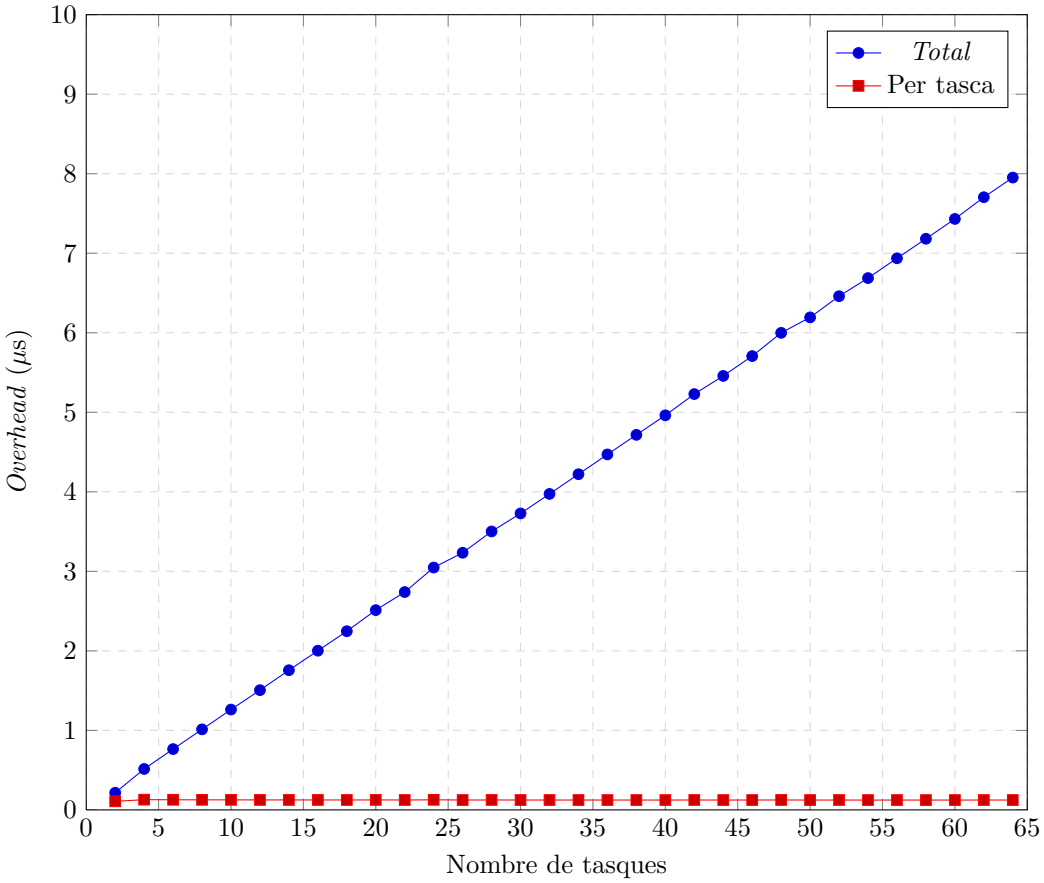

Figura 4: *Overhead* total i per tasca en funció del nombre de tasques.