Grai2º curso / 2º cuatr.

Grado Ing. Inform.

Doble Grado Ing. Inform. y Mat.

Arquitectura de Computadores (AC)

Cuaderno de prácticas.
Bloque Práctico 2. Programación paralela II: Cláusulas OpenMP

Estudiante (nombre y apellidos): Carlos Sánchez Páez
Grupo de prácticas: A2
Fecha de entrega: 16/04/2018
Fecha evaluación en clase: 17/04/2018

Ejercicios basados en los ejemplos del seminario práctico

1. ¿Qué ocurre si en el ejemplo del seminario `shared-clause.c` se añade a la directiva `parallel` la cláusula `default(none)`? (añada una captura de pantalla que muestre lo que ocurre) **(b)** Resuelva el problema generado sin eliminar `default(none)`.Añada el código con la modificación al cuaderno de prácticas.

RESPUESTA:
La cláusula default(none) obliga al programador a indicar el ámbito de las variables. Como todas están declaradas antes de la directiva, no son reconocidas. El arreglo es indicar la directiva al principio del programa.

CAPTURA CÓDIGO FUENTE: `shared-clauseModificado.c`

```
int main() {

    #pragma omp parallel default(none)
    {
        int i, n = 7;
        int a[n];
        for (i = 0; i < n; i++)
            a[i] = i + 1;
        #pragma omp parallel for shared(a)
        for (i = 0; i < n; i++)
            a[i] += i;

    printf("Después de parallel for:\n");

    for (i = 0; i < n; i++)
        printf("a[%d]=%d\n", i, a[i]);
    }

}
```

CAPTURAS DE PANTALLA:

```
#pragma omp parallel default(none)
{
    #pragma omp parallel for shared(a)
    for (i = 0; i < n; i++)
        a[i] += i;
}
printf("Después de parallel for:\n");
```

2. ¿Qué ocurre si en `private-clause.c` se inicializa la variable `suma` fuera de la construcción `parallel` en lugar de dentro? (inicialice suma a un valor distinto de 0 dentro y fuera de `parallel`) Razone su respuesta. Añada el código con la modificación al cuaderno de prácticas.

RESPUESTA:

Que el valor inicial de suma queda indefinido, por lo que el resultado es erróneo.

CAPTURA CÓDIGO FUENTE: `private-clauseModificado.c`

```
suma = 1;
#pragma omp parallel private(suma)
{
    #pragma omp for
    for (i = 0; i < n; i++) {
        suma = suma + a[i];
        printf("thread %d suma a[%d]/", omp_get_thread_num(), i);
    }
    printf("\n* thread %d suma=%d", omp_get_thread_num(), suma);
}
printf("\n");
```

CAPTURAS DE PANTALLA:

```
thread 0 suma a[0]/thread 6 suma a[6]/thread 5 suma a[5]/thread 1 suma a[1]/thre
ad 3 suma a[3]/thread 2 suma a[2]/thread 4 suma a[4]/
* thread 0 suma=100432328
* thread 5 suma=6
* thread 6 suma=7
* thread 1 suma=2
* thread 7 suma=0
* thread 2 suma=3
* thread 3 suma=4
* thread 4 suma=5
[Carlos Sanchez Paez csp98@csp98-GL552VW:~/Escritorio/AC/Prácticas/practica2/src
```

3. ¿Qué ocurre si en `private-clause.c` se elimina la cláusula `private(suma)`? ¿A qué cree que es debido?

RESPUESTA:

Como la variable pasa a ser compartida, vale lo mismo para todas las hebras.

CAPTURA CÓDIGO FUENTE: `private-clauseModificado3.c`

```
#pragma omp parallel
{
    suma = 0;
    #pragma omp for
    for (i = 0; i < n; i++) {
        suma = suma + a[i];
        printf("thread %d suma a[%d]/", omp_get_thread_num(), i);
    }
    printf("\n* thread %d suma=%d", omp_get_thread_num(), suma);
}
printf("\n");
```

CAPTURAS DE PANTALLA:

```
[Carlos Sanchez Paez csp98@csp98-GL552VW:~/Escritorio/AC/Prácticas/practica2/src
] 2018-04-10 martes
$./private_clause_modificado2
thread 0 suma a[0]/thread 6 suma a[6]/thread 3 suma a[3]/thread 4 suma a[4]/thre
ad 1 suma a[1]/thread 2 suma a[2]/thread 5 suma a[5]/
* thread 6 suma=7
* thread 0 suma=7
* thread 3 suma=7
* thread 4 suma=7
* thread 1 suma=7
* thread 7 suma=7
* thread 5 suma=7
* thread 2 suma=7
```

4. En la ejecución de `firstlastprivate.c` de la pag. 21 del seminario se imprime un 6 fuera de la región `parallel`. ¿El código imprime siempre 6 fuera de la región `parallel`? Razone su respuesta.

RESPUESTA:

Tendrá el primer valor que se modifique en la construcción.

5. ¿Qué se observa en los resultados de ejecución de `copyprivate-clause.c` cuando se elimina la cláusula `copyprivate(a)` en la directiva `single`? ¿A qué cree que es debido?

RESPUESTA:

Como la variable no se difunde a las otras hebras, tiene un valor inicial distinto para cada una, siendo el resultado incorrecto.

CAPTURA CÓDIGO FUENTE: `copyprivate-clauseModificado.c`

```
#pragma omp parallel
{
    int a;
    #pragma omp single
    {
        printf("\nIntroduce valor de inicialización a:");
        scanf("%d",&a);
        printf("\nSingle ejecutada por el thread %d\n",omp_get_thread_num());
    }
    #pragma omp for
    for(i=0;i<n;i++)
        b[i]=a;
    printf("Después de la región parallel:\n");
    for(i=0;i<n;i++)
        printf("b[%d]=%d\t",i,b[i]);
    printf("\n");

}
```

CAPTURAS DE PANTALLA:

```
$./copyprivate_clause_modificado

Introduce valor de inicialización a:1

Single ejecutada por el thread 4
Después de la región parallel:
b[0]=0  b[1]=0  b[2]=0  b[3]=0  b[4]=0  b[5]=1  b[6]=0  b[7]=0  b[8]=0
Después de la región parallel:
b[0]=0  b[1]=0  b[2]=0  b[3]=0  b[4]=0  b[5]=1  b[6]=0  b[7]=0  b[8]=0
Después de la región parallel:
b[0]=0  b[1]=0  b[2]=0  b[3]=0  b[4]=0  b[5]=1  b[6]=0  b[7]=0  b[8]=0
Después de la región parallel:
b[0]=0  b[1]=0  b[2]=0  b[3]=0  b[4]=0  b[5]=1  b[6]=0  b[7]=0  Después de la re
gión parallel:
b[0]=0  b[1]=0  b[2]=0  b[3]=0  b[4]=0  b[5]=1  b[6]=0  Después de la región par
allel:
b[0]=0  b[1]=0  b[2]=0  b[3]=0  b[4]=0  b[5]=1  b[6]=0  b[7]=0  b[8]=0
b[8]=0
b[7]=0
Después de la región parallel:
b[0]=0  b[1]=0  b[2]=0  b[3]=0  b[4]=0  b[5]=1  b[6]=0  b[7]=0  b[8]=0
Después de la región parallel:
b[0]=0  b[1]=0  b[2]=0  b[3]=0  b[4]=0  b[5]=1  b[6]=0  b[7]=0  b[8]=0
[Carlos Sanchez Paez csp98@csp98-GL552VW:~/Escritorio/AC/Prácticas/practica2/src
] 2018-04-10 martes
$
```

6. En el ejemplo `reduction-clause.c` sustituya `suma=0` por `suma=10`. ¿Qué resultado se imprime ahora? Justifique el resultado

RESPUESTA:

Todos los resultados son 10 unidades mayores, ya que el valor inicial de la suma se incrementa en 10.

CAPTURA CÓDIGO FUENTE: `reduction-clauseModificado.c`

```c
int main(int argc,char **argv) {
    int i,n=20,a[n],suma=10;
    if(argc<2){
        fprintf(stderr, "Falta iteraciones\n");
        exit(-1);
    }
    n=atoi(argv[1]);
    if(n>20){
        n=20;
        printf("n=%d",n);
    }
    for(i=0;i<n;i++)
        a[i]=i;
    #pragma omp parallel for reduction(+:suma)
    for(i=0;i<n;i++)
        suma+=a[i];
    printf("Tras 'parallel' suma=%d\n",suma);


}
```

CAPTURAS DE PANTALLA:

```
[Carlos Sanchez Paez csp98@csp98-GL552VW:~/Escritorio/AC/Prácticas/practica2/src
] 2018-04-10 martes
$./reduction_clause_modificado 10
Tras 'parallel' suma=55
[Carlos Sanchez Paez csp98@csp98-GL552VW:~/Escritorio/AC/Prácticas/practica2/src
] 2018-04-10 martes
```

7. En el ejemplo `reduction-clause.c`, elimine `reduction()` de `#pragma omp parallel for reduction(+:suma)` y haga las modificaciones necesarias para que se siga realizando la suma de los componentes del vector `a` en paralelo sin usar directivas de trabajo compartido .

RESPUESTA:

CAPTURA CÓDIGO FUENTE: `reduction-clauseModificado7.c`

```c
#pragma omp parallel for
for (i = 0; i < n; i++)
    #pragma omp critical
    suma += a[i];
printf("Tras 'parallel' suma=%d\n", suma);
```

CAPTURAS DE PANTALLA:

```
[CarlosSánchezPáez csp98@csp98-Sobremesa-Linux:~/Desktop/AC/Prácticas/practica2/
src] 2018-04-15 domingo
$gcc -o reduction_clause_modificado2 reduction_clause_modificado2.c -fopenmp
[CarlosSánchezPáez csp98@csp98-Sobremesa-Linux:~/Desktop/AC/Prácticas/practica2/
src] 2018-04-15 domingo
$./reduction_clause_modificado2 50
n=20Tras 'parallel' suma=190
[CarlosSánchezPáez csp98@csp98-Sobremesa-Linux:~/Desktop/AC/Prácticas/practica2/
src] 2018-04-15 domingo
$./reduction_clause_modificado2 50
n=20Tras 'parallel' suma=190
[CarlosSánchezPáez csp98@csp98-Sobremesa-Linux:~/Desktop/AC/Prácticas/practica2/
src] 2018-04-15 domingo
$./reduction_clause_modificado2 50
n=20Tras 'parallel' suma=190
[CarlosSánchezPáez csp98@csp98-Sobremesa-Linux:~/Desktop/AC/Prácticas/practica2/
src] 2018-04-15 domingo
$./reduction_clause_modificado2 50
n=20Tras 'parallel' suma=190
[CarlosSánchezPáez csp98@csp98-Sobremesa-Linux:~/Desktop/AC/Prácticas/practica2/
src] 2018-04-15 domingo
$
```

Resto de ejercicios

8. Implementar un programa secuencial en C que calcule el producto de una matriz cuadrada, M, por un vector, v1 (implemente una versión para variables globales y otra para variables dinámicas, use una de estas versiones en los siguientes ejercicios):

$$v2 = M \bullet v1;\; v2(i) = \sum_{k=0}^{N-1} M(i,k) \bullet v(k),\; i = 0,\ldots N-1$$

NOTAS: (1) el número de filas /columnas N de la matriz deben ser argumentos de entrada al programa; (2) se debe inicializar la matriz y el vector antes del cálculo; (3) se debe asegurar que el programa calcula la suma correctamente imprimiendo todos los componentes del vector resultante, v3, para tamaños pequeños de los vectores (por ejemplo, N = 8 y N=11); (5) se debe imprimir sea cual sea el tamaño de los vectores el tiempo de ejecución del código paralelo que calcula el producto matriz vector y, al menos, el primer y último componente del resultado (esto último evita que las optimizaciones del compilador eliminen el código de la suma).

CAPTURA CÓDIGO FUENTE: `pmv-secuencial.c`

```c
/***Producto***/
double temp;
clock_gettime(CLOCK_REALTIME, &cgt1);
for (int i = 0; i < TAM; i++) {
    temp = 0;
    for (int j = 0; j < TAM; j++) {
        temp += m[i][j] * v1[j];
    }
    v2[i] = temp;
}
clock_gettime(CLOCK_REALTIME, &cgt2);
ncgt = (double) (cgt2.tv_sec - cgt1.tv_sec) + (double) ((cgt2.tv_nsec - cgt1.tv_nsec) / (1.e+9));
/*************/
```

CAPTURAS DE PANTALLA:

```
[CarlosSánchezPáez csp98@csp98-Sobremesa-Linux:~/Desktop/AC/Prácticas/practica2/
src] 2018-04-15 domingo
$./pmv_secuencial 4
---------Valores iniciales------
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000
v1[0]=4.000000
v1[1]=4.000000
v1[2]=4.000000
v1[3]=4.000000
--------------------------------
---------Valores finales------
v2[0]=4.000000
v2[1]=4.000000
v2[2]=4.000000
v2[3]=4.000000
--------------------------------

Tiempo (seg): 0.000000277        Tamaño: 4      v2[0]=4.000000  v[3]=4.000000
[CarlosSánchezPáez csp98@csp98-Sobremesa-Linux:~/Desktop/AC/Prácticas/practica2/
src] 2018-04-15 domingo
$
```

9. Implementar en paralelo el producto matriz por vector con OpenMP a partir del código escrito en el ejercicio anterior usando la directiva `for` . Debe implementar dos versiones del código (consulte la lección 5/Tema 2):

    a. una primera que paralelice el bucle que recorre las filas de la matriz y

    b. una segunda que paralelice el bucle que recorre las columnas.

    Use las directivas que estime oportunas y las cláusulas que sean necesarias **excepto la cláusula `reduction`**. Se debe paralelizar también la inicialización de las matrices. Respecto a este ejercicio:

    - Anote en su cuaderno de prácticas todos los errores de compilación que se han generado durante la realización del ejercicio y explique cómo los ha resuelto (especifique qué ayudas externas ha usado o recibido).
    - Anote todos los errores en tiempo de ejecución que se han generado durante la realización del ejercicio y explique cómo los ha resuelto (especifique qué ayudas externas ha usado o recibido).

    NOTAS: (1) el número de filas /columnas N de la matriz deben ser argumentos de entrada; (2) se debe inicializar la matriz y el vector antes del cálculo; (3) se debe asegurar que el programa calcula la suma correctamente imprimiendo todos los componentes del vector resultante, v3, para tamaños pequeños de los vectores (por ejemplo, N = 8 y N=11); (5) se debe imprimir sea cual sea el tamaño de los vectores el tiempo de ejecución del código que calcula el producto matriz vector y, al menos, el primer y último componente del resultado (esto último evita que las optimizaciones del compilador eliminen el código de la suma).

CAPTURA CÓDIGO FUENTE : `pmv-OpenMP-a.c`

```c
/***Producto***/
double temp;
clock_gettime(CLOCK_REALTIME, &cgt1);
#pragma omp parallel for private(temp)
for (int i = 0; i < TAM; i++) {
    temp = 0;
    for (int j = 0; j < TAM; j++) {
        temp += m[i][j] * v1[j];
    }
    v2[i] = temp;
}
clock_gettime(CLOCK_REALTIME, &cgt2);
ncgt = (double) (cgt2.tv_sec - cgt1.tv_sec) + (double) ((cgt2.tv_nsec - cgt1.tv_nsec) / (1.e+9));
/*************/
```

CAPTURA CÓDIGO FUENTE: `pmv-OpenMP-b.c`

```
/***Producto***/
double temp;
clock_gettime(CLOCK_REALTIME, &cgt1);
for (int i = 0; i < TAM; i++) {
    temp = 0;
    #pragma omp parallel for
    for (int j = 0; j < TAM; j++) {
        #pragma omp critical
        temp += m[i][j] * v1[j];
    }
    v2[i] = temp;
}
clock_gettime(CLOCK_REALTIME, &cgt2);
ncgt = (double) (cgt2.tv_sec - cgt1.tv_sec) + (double) ((cgt2.tv_nsec - cgt1.tv_nsec) / (1.e+9));
/*************/
```

RESPUESTA:

En la segunda versión olvidé añadir *critical*, por lo que algunos valores eran erróneos.

CAPTURAS DE PANTALLA:

```
[CarlosSánchezPáez csp98@csp98-Sobremesa-Linux:~/Desktop/AC/Prácticas/practica2/
src] 2018-04-15 domingo
$./pmv_openMP_a 4
---------Valores iniciales------
1.000000 0.000000 0.000000 0.000000
0.000000 1.000000 0.000000 0.000000
0.000000 0.000000 1.000000 0.000000
0.000000 0.000000 0.000000 1.000000
v1[0]=4.000000
v1[1]=4.000000
v1[2]=4.000000
v1[3]=4.000000
--------------------------------
---------Valores finales------
v2[0]=4.000000
v2[1]=4.000000
v2[2]=4.000000
v2[3]=4.000000
--------------------------------

Tiempo (seg): 0.000003714        Tamaño: 4      v2[0]=4.000000  v[3]=4.000000
[CarlosSánchezPáez csp98@csp98-Sobremesa-Linux:~/Desktop/AC/Prácticas/practica2/
src] 2018-04-15 domingo
$
```

```
[CarlosSánchezPáez csp98@csp98-Sobremesa-Linux:~/Desktop/AC/Prácticas/practica2/
src] 2018-04-15 domingo
$./pmv_openMP_b 4
--------Valores iniciales------
1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000
v1[0]=1.000000
v1[1]=1.000000
v1[2]=1.000000
v1[3]=1.000000
--------------------------------
--------Valores finales------
v2[0]=4.000000
v2[1]=4.000000
v2[2]=4.000000
v2[3]=4.000000
--------------------------------

Tiempo (seg): 0.000017394        Tamaño: 4      v2[0]=4.000000  v[3]=4.000000
[CarlosSánchezPáez csp98@csp98-Sobremesa-Linux:~/Desktop/AC/Prácticas/practica2/
src] 2018-04-15 domingo
$
```

10. A partir de la segunda versión de código paralelo desarrollado en el ejercicio anterior, implementar una versión paralela del producto matriz por vector con OpenMP que use para comunicación/sincronización la cláusula `reduction`. Respecto a este ejercicio:
    - Anote en su cuaderno de prácticas todos los errores de compilación que se han generado durante la realización del ejercicio y explique cómo los ha resuelto (especifique qué ayudas externas ha usado o recibido).
    - Anote todos los errores en tiempo de ejecución que se han generado durante la realización del ejercicio y explique cómo los ha resuelto (especifique qué ayudas externas ha usado o recibido).

CAPTURA CÓDIGO FUENTE: `pmv-OpenmMP-reduction.c`

```
/***Producto***/
double temp;
clock_gettime(CLOCK_REALTIME, &cgt1);
for (int i = 0; i < TAM; i++) {
    temp = 0;
    #pragma omp parallel for reduction(+:temp)
    for (int j = 0; j < TAM; j++) {
        temp += m[i][j] * v1[j];
    }
    v2[i] = temp;
}
clock_gettime(CLOCK_REALTIME, &cgt2);
ncgt = (double) (cgt2.tv_sec - cgt1.tv_sec) + (double) ((cgt2.tv_nsec - cgt1.tv_nsec) / (1.e+9));
/*************/
```

RESPUESTA:

No he obtenido errores.

CAPTURAS DE PANTALLA:

```
[CarlosSánchezPáez csp98@csp98-Sobremesa-Linux:~/Desktop/AC/Prácticas/practica2/
src] 2018-04-14 sábado
$./pmv_openMP_reduction 2
---------Valores iniciales------
1.000000 1.000000
1.000000 1.000000
v1[0]=1.000000
v1[1]=1.000000
--------------------------------
---------Valores finales------
v2[0]=2.000000
v2[1]=2.000000
--------------------------------

Tiempo (seg): 0.000008084        Tamaño: 2      v2[0]=2.000000  v[1]=2.000000
[CarlosSánchezPáez csp98@csp98-Sobremesa-Linux:~/Desktop/AC/Prácticas/practica2/
src] 2018-04-14 sábado
$11
```

11. Ayudándose de una hoja de cálculo (recuerde que en las aulas está instalado OpenOffice) realice una tabla y una gráfica que permitan comparar la escalabilidad (ganancia en velocidad en función del número de cores) en atcgrid y en su PC del mejor código paralelo de los tres implementados en los ejercicios anteriores para dos tamaños (N) distintos (consulte la Lección 6/Tema 2). Usar –O2 al compilar. Justificar por qué el código escogido es el mejor. NOTA: Nunca ejecute en atcgrid código que imprima todos los componentes del resultado.

CAPTURAS DE PANTALLA (que justifique el código elegido):

TABLA Y GRÁFICA (por *ejemplo* para 1-4 threads PC local, y para 1-12 threads en atcgrid, tamaños-N-: un N entre 30000 y 100000, y otro entre 5000 y 30000):

FILAS ATCGRID		
Hebras	Tamaño 50000	Tamaño 15000
1	3.793917973	0.325837249
2	3.068337155	0.181156315
3	4.821470553	0.135106676
4	1.644303041	0.115471731
5	1.354134296	0.090380333
6	1.369119680	0.097650344
7	1.270313763	0.078850222
8	1.154179603	0.080339775
9	1.022370905	0.084580946
10	1.010759934	0.078341100
11	1.167183398	0.084720099
12	1.492167260	0.083414653

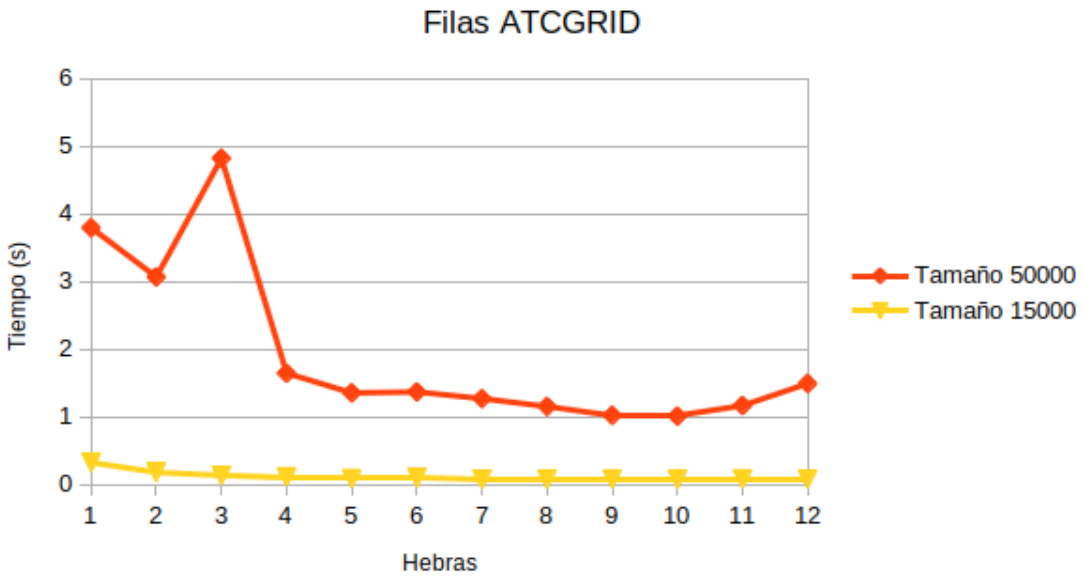
COLUMNAS ATCGRID		
Hebras	Tamaño 50000	Tamaño 15000
1	3.851304782	0.368914780
2	3.244618799	0.234287427
3	7.250571502	0.207500384
4	4.157084064	0.188826578
5	4.478545865	0.194309804
6	4.291524103	0.187023037
7	4.226857248	0.190970625
8	4.108297130	0.205222779
9	4.191943849	0.218790251
10	4,340795647	0.233664199
11	4.826995498	0.241837089
12	4.412253249	0.255314870

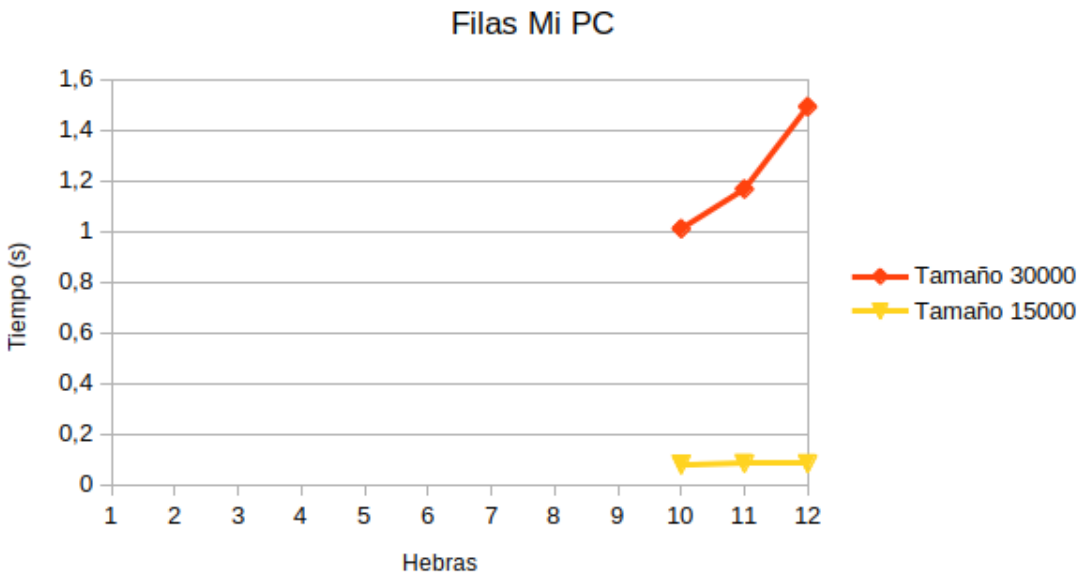
REDUCTION ATCGRID		
Hebras	Tamaño 50000	Tamaño 15000
1	4.124911167	0.389901129
2	3.574643857	0.209502490
3	6.449126724	0.176096364
4	4.130098301	0.170864313
5	4.223723927	0.159540085
6	3.481909173	0.153685095
7	2.041934040	0.157139145
8	3.691784114	0.157405427
9	3.803895033	0.157387529
10	3.871589255	0.159120859
11	3.926360353	0.174705089
12	4.411692152	0.173980182

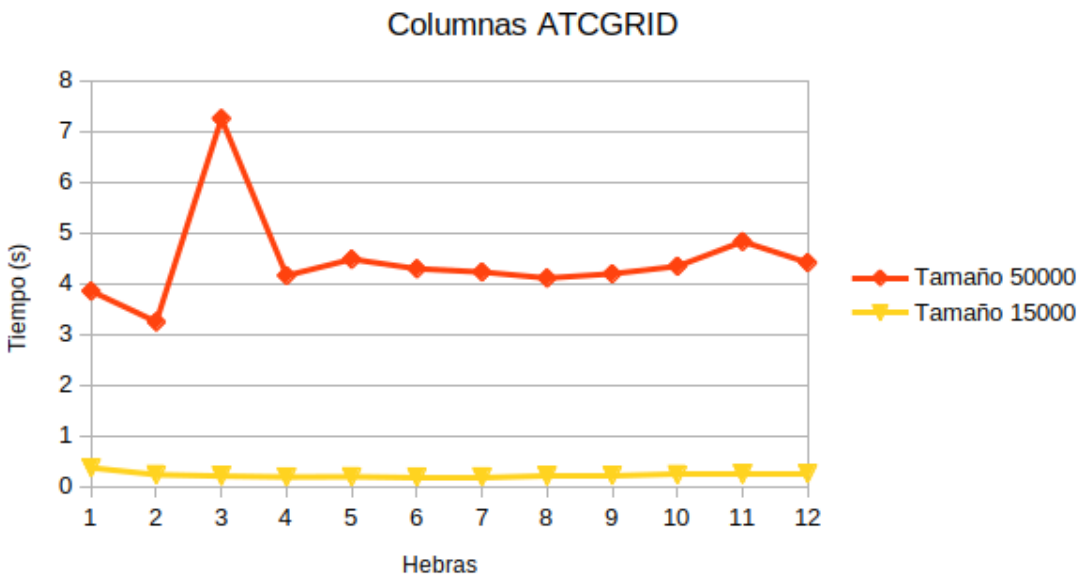
COLUMNAS MI PC		
Hebras	Tamaño 30000	Tamaño 15000
1	1.535096024	0.382413347
2	0.821688634	0.219800022
3	0.625777589	0.184601541
4	0.565926197	0.182920727
5	0.583000104	0.192400454
6	0.587091721	0.201968112
7	0.604705890	0.223200933
8	0.708411463	0.376962094

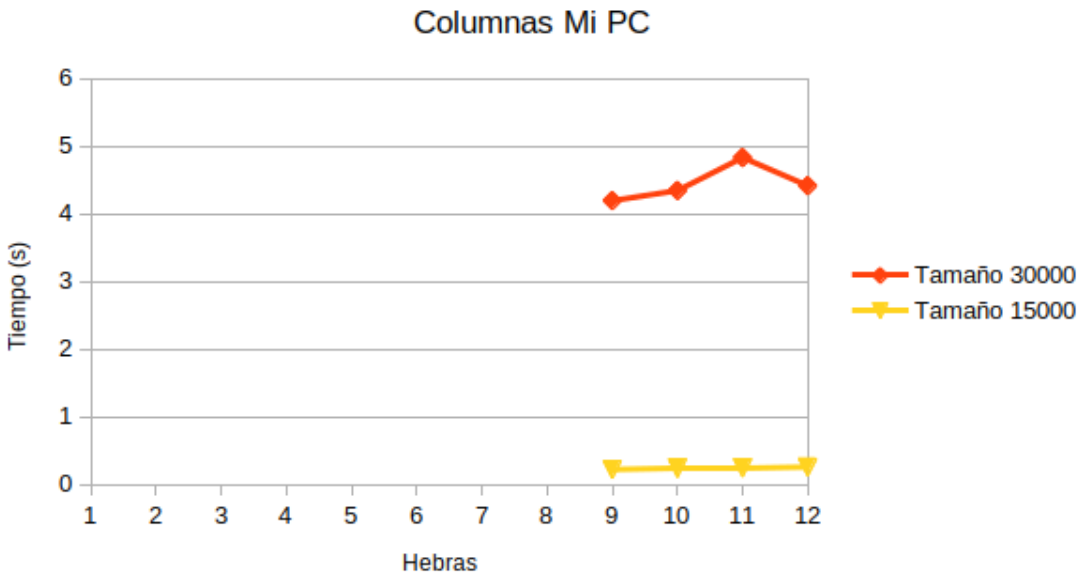
FILAS MI PC		
Hebras	Tamaño 30000	Tamaño 15000
1	1.485576399	0.370272011
2	0.785878909	0.190978261
3	0.534578991	0.131571456
4	0.434392372	0.107256948
5	0.407551516	0.105028519
6	0.376822791	0.099576320
7	0.347345816	0.092166799
8	0.339597641	0.092203399

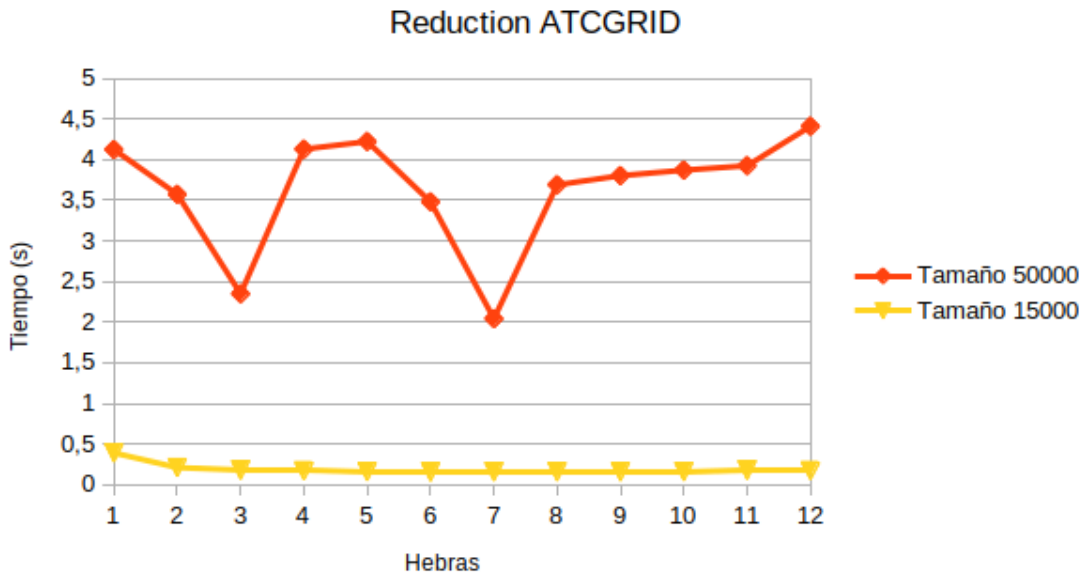
REDUCTION MI PC		
Hebras	Tamaño 30000	Tamaño 15000
1	1.493607898	0.367356980
2	0.789514398	0.205467996
3	0.555049999	0.155382286
4	0.467963813	0.135957072
5	0.474619406	0.139341915
6	0.446742206	0.142157193
7	0.443722923	0.155302536
8	0.683743176	0.218992996

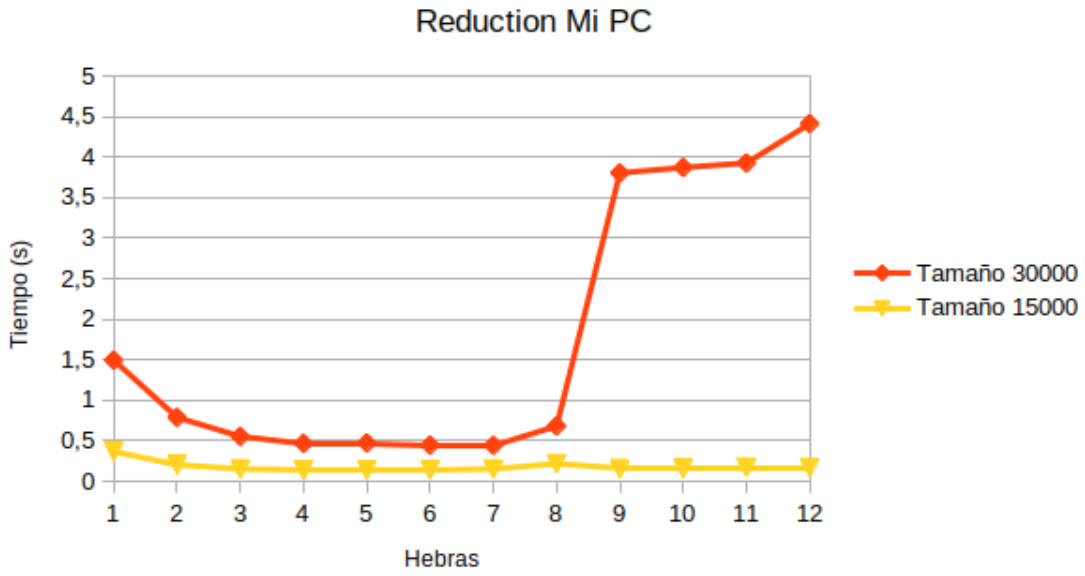

Filas ATCGRID
Tiempo (s)
0
1
2
3
4
5
6
1
2
3
4
5
6
7
8
9
10
11
12
Hebras
Tamaño 50000
Tamaño 15000


Filas Mi PC
Tiempo (s)
0
0,2
0,4
0,6
0,8
1
1,2
1,4
1,6
1
2
3
4
5
6
7
8
9
10
11
12
Hebras
Tamaño 30000
Tamaño 15000


Columnas ATCGRID
Tiempo (s)
Hebras
Tamaño 50000
Tamaño 15000


Columnas Mi PC
Tiempo (s)
Hebras
Tamaño 30000
Tamaño 15000


Reduction ATCGRID
Tiempo (s)
5
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
1 2 3 4 5 6 7 8 9 10 11 12
Hebras
Tamaño 50000
Tamaño 15000


Reduction Mi PC
Tiempo (s)
5
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
1 2 3 4 5 6 7 8 9 10 11 12
Hebras
Tamaño 30000
Tamaño 15000

COMENTARIOS SOBRE LOS RESULTADOS:

Podemos ver que llega un punto en ambos sistemas en el que el tiempo de procesamiento crece a mayor número de hebras. Esto se produce cuando el *overhead* es tan grande que hace que la ganancia sea nula.

También podemos apreciar la potencia del nodo de cómputo. Mientras que en mi PC con tamaños de 50.000 elementos tardaba aproximadamente 5 minutos, *atcgrid* lo consigue en pocos segundos.