

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): Marta Gómez Macías

Grupo de prácticas: C1

Fecha de entrega: 21/04/2015 - 23:59

Fecha evaluación en clase: 22/04/2015 - 15:30

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: Nos devuelve un error porque no hemos especificado el ámbito de la variable n . Para resolverlo, añadimos la variable n a la lista de variables compartidas.

CÓDIGO FUENTE: shared-clauseModificado.c

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

CAPTURAS DE PANTALLA:

```
[marta@marta-PC Ejercicio1]$ gcc -fopenmp -O2 shared-clause.c -o shared_a
shared-clause.c: En la función 'main':
shared-clause.c:14:13: error: no se especificó 'n' en el paralelo que lo contiene
#pragma omp parallel for default(none) shared(a)
      ^
shared-clause.c:14:13: error: paralelo contenedor
[marta@marta-PC Ejercicio1]$
```

2. ¿Qué ocurre si en private-clause.c se inicializa la variable suma fuera de la construcción parallel en lugar de dentro? Razone su respuesta. Añada el código con la modificación al cuaderno de prácticas.

RESPUESTA: Obtenemos unos resultados de suma erróneos porque la variable suma toma un valor desconocido al entrar en la región parallel.

CÓDIGO FUENTE: private-clauseModificado.c

```
suma = 0;
#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);
}
```

CAPTURAS DE PANTALLA:

```
[marta@marta-PC Ejercicio2]$ gcc -fopenmp -O2 private-clauseModificado.c -o private
[marta@marta-PC Ejercicio2]$ ./private
thread 2 suma a[4] / thread 2 suma a[5] / thread 1 suma a[2] / thread 1 suma a[3] / thread 3 suma a[6] / thread 0 suma a[0] / thread 0 suma a[1] /
*thread 2 suma = 4196361
*thread 3 suma = 4196358
*thread 1 suma = 4196357
*thread 0 suma = 5
[marta@marta-PC Ejercicio2]$
```

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

RESPUESTA: Obtenemos un resultado incorrecto porque todas las hebras trabajan sobre la misma variable `suma`, ya que al ser definida fuera de la región `parallel` es una variable compartida.

CÓDIGO FUENTE: `private-clauseModificado3.c`

```
#pragma omp parallel /*private(suma)*/
{
    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);
}
```

CAPTURAS DE PANTALLA:

```
[marta@marta-PC Ejercicio3]$ gcc -fopenmp -O2 private-clauseModificado3.c -o private
[marta@marta-PC Ejercicio3]$ ./private
thread 3 suma a[6] / thread 0 suma a[0] / thread 0 suma a[1] / thread 2 suma a[4] / thread 2 suma a[5] / thread 1 suma a[2] / thread 1 suma a[3] /
*thread 3 suma = 15
*thread 1 suma = 15
*thread 0 suma = 15
*thread 2 suma = 15
[marta@marta-PC Ejercicio3]$
```

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: Sí, porque `lastprivate` hace que `suma` se quede con el valor del último thread, el 3. Al ser siempre el valor del thread 3 el 6, siempre imprimirá dicho valor.

CAPTURAS DE PANTALLA:

```

[marta@marta-PC Ejercicio4]$ ./firstlastprivate-clause
thread 0 suma a[0] suma=0
thread 0 suma a[1] suma=1
thread 1 suma a[2] suma=2
thread 1 suma a[3] suma=5
thread 2 suma a[4] suma=4
thread 2 suma a[5] suma=9
thread 3 suma a[6] suma=6
Fuera de la construccion parallel suma=6
[marta@marta-PC Ejercicio4]$ ./firstlastprivate-clause
thread 0 suma a[0] suma=0
thread 0 suma a[1] suma=1
thread 1 suma a[2] suma=2
thread 1 suma a[3] suma=5
thread 2 suma a[4] suma=4
thread 2 suma a[5] suma=9
thread 3 suma a[6] suma=6
Fuera de la construccion parallel suma=6
[marta@marta-PC Ejercicio4]$ ./firstlastprivate-clause
thread 0 suma a[0] suma=0
thread 0 suma a[1] suma=1
thread 2 suma a[4] suma=4
thread 2 suma a[5] suma=9
thread 1 suma a[2] suma=2
thread 1 suma a[3] suma=5
thread 3 suma a[6] suma=6
Fuera de la construccion parallel suma=6
[marta@marta-PC Ejercicio4]$ ./firstlastprivate-clause
thread 1 suma a[2] suma=2
thread 1 suma a[3] suma=5
thread 0 suma a[0] suma=0
thread 0 suma a[1] suma=1
thread 2 suma a[4] suma=4
thread 2 suma a[5] suma=9
thread 3 suma a[6] suma=6
Fuera de la construccion parallel suma=6
[marta@marta-PC Ejercicio4]$ ./firstlastprivate-clause
thread 2 suma a[4] suma=4
thread 2 suma a[5] suma=9
thread 3 suma a[6] suma=6
thread 1 suma a[2] suma=2
thread 1 suma a[3] suma=5
thread 0 suma a[0] suma=0
thread 0 suma a[1] suma=1
Fuera de la construccion parallel suma=6
[marta@marta-PC Ejercicio4]$

```

5. ¿Qué ocurre si en copyprivate-clause.c se elimina la cláusula copyprivate(a) en la directiva single? ¿A qué cree que es debido?

RESPUESTA: Que la difusión del valor introducido por teclado no se hace y, por tanto, en vez de inicializar todo el vector b con el valor introducido por teclado, se inicializa sólo la parte del thread que ejecutó el single y el resto se inicializa a un valor indeterminado.

CÓDIGO FUENTE: copyprivate-clauseModificado.c

```

#pragma omp parallel
{
    int a;
    #pragma omp single /*copyprivate(a)*/
    {
        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;
}

```

CAPTURAS DE PANTALLA:

```
[marta@marta-PC Ejercicio5]$ gcc -O2 -fopenmp -o copyprivate copyprivate-clauseModificado.c
[marta@marta-PC Ejercicio5]$ ./copyprivate

Introduce valor de inicialización a:3

Single ejecutada por el thread 0
Después de la región parallel:
b[0] = 3      b[1] = 3      b[2] = 3      b[3] = 0      b[4] = 0      b[5] = 0      b[6] = 0      b[7] = 0      b[8] = 0
[marta@marta-PC Ejercicio5]$
```

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

RESPUESTA: El resultado final de la suma es 10 unidades mayor. Esto se debe a que la cláusula reduction, después de sumar todos los valores de las iteraciones, se los suma al valor original de la variable suma. Por eso, el resultado es la suma de las iteraciones del bucle for y el valor original que tuviera suma. Si fuese 20, el valor sería 20 unidades mayor y así con cualquier número.

CÓDIGO FUENTE: reduction-clauseModificado.c

```
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", 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];
```

CAPTURAS DE PANTALLA:

```
[marta@marta-PC Ejercicio6]$ gcc -O2 -fopenmp -o reduction-suma0 reduction-clause.c
[marta@marta-PC Ejercicio6]$ gcc -O2 -fopenmp -o reduction-suma10 reduction-clause.c
[marta@marta-PC Ejercicio6]$ ./reduction-suma0 10
Tras 'parallel' suma = 45
[marta@marta-PC Ejercicio6]$ ./reduction-suma10 10
Tras 'parallel' suma = 55
[marta@marta-PC Ejercicio6]$ ./reduction-suma10 20
Tras 'parallel' suma = 200
[marta@marta-PC Ejercicio6]$ ./reduction-suma0 20
Tras 'parallel' suma = 190
[marta@marta-PC Ejercicio6]$
```

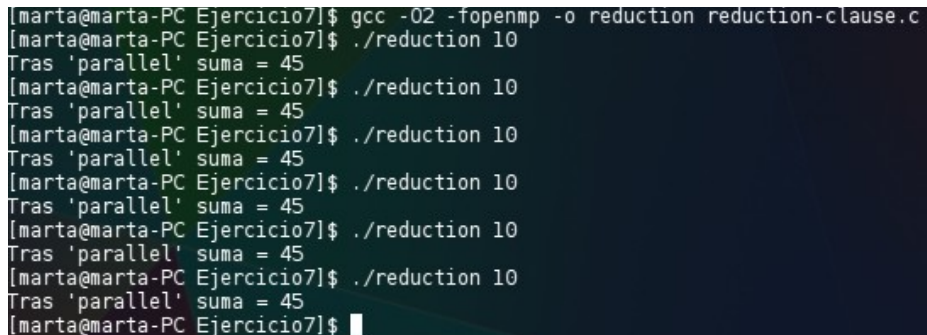
7. En el ejemplo reduction-clause.c, elimine reduction(+:suma) de #pragma omp parallel for reduction(+:suma) y haga las modificaciones necesarias para que se siga realizando la suma de los componentes del vector a en paralelo.

RESPUESTA:

CÓDIGO FUENTE: reduction-clauseModificado7.c

```
#pragma omp parallel firstprivate(suma_pri)
{
    #pragma omp for
    for(i = 0; i < n; i++)
        suma_pri += a[i];

    #pragma omp atomic
    suma_final += suma_pri;
}
```

CAPTURAS DE PANTALLA:


```
[marta@marta-PC Ejercicio7]$ gcc -O2 -fopenmp -o reduction reduction-clause.c
[marta@marta-PC Ejercicio7]$ ./reduction 10
Tras 'parallel' suma = 45
[marta@marta-PC Ejercicio7]$ ./reduction 10
Tras 'parallel' suma = 45
[marta@marta-PC Ejercicio7]$ ./reduction 10
Tras 'parallel' suma = 45
[marta@marta-PC Ejercicio7]$ ./reduction 10
Tras 'parallel' suma = 45
[marta@marta-PC Ejercicio7]$ ./reduction 10
Tras 'parallel' suma = 45
[marta@marta-PC Ejercicio7]$ ./reduction 10
Tras 'parallel' suma = 45
[marta@marta-PC Ejercicio7]$
```

8. Implementar un programa secuencial en C que calcule el producto de una matriz cuadrada, M, por un vector, v1:

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

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).

CÓDIGO FUENTE: pmv-secuencial.c

```
for (i=0; i<N;i++) {
    for (k=0; k<N; k++)
        suma+=matriz[k][i];

    vector_resultado[i] = suma * vector[i];
    sum si estás por la escuela puedes pasarte esta misma mañana.

Saludosa = 0;
}
```

CAPTURAS DE PANTALLA:


```

[marta@marta-PC Ejercicio8]$ gcc -O2 -o pmvs pmv-secuencial.c
[marta@marta-PC Ejercicio8]$ ./pmvs 3
Tiempo: 0.000000511
v[0] = 18
v[1] = 18
v[2] = 18
[marta@marta-PC Ejercicio8]$ ./pmvs 10
Tiempo: 0.000000758
v[0] = 60
v[1] = 60
v[2] = 60
v[3] = 60
v[4] = 60
v[5] = 60
v[6] = 60
v[7] = 60
v[8] = 60
v[9] = 60
[marta@marta-PC Ejercicio8]$ ./pmvs 12
Tiempo: 0.000001223    v[0] = 72    v[11] = 72
[marta@marta-PC Ejercicio8]$

```

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):

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

CÓDIGO FUENTE : pmv-OpenMP-a.c

```

#pragma omp parallel for private(i,j)
for (i = 0; i<N; i++) {
    for (j = 0; j<N; j++)
        matriz[i][j] = 3;
    vector[i] = 2;
}
int vector_resultado[N], k = 0, suma=0;

```

```
double start, end, tiempo;
start = omp_get_wtime();

#pragma omp parallel for private(i,k,suma)
for (i=0; i<N; i++) {
    suma = 0;
    for (k=0; k<N; k++)
        suma+=matriz[k][i];
    vector_resultado[i] = suma * vector[i];
}
end = omp_get_wtime();
tiempo = end - start;
```

CÓDIGO FUENTE: pmv-OpenMP-b.c

```
for (i=0; i<N; i++) {
    suma = 0;
    #pragma omp parallel firstprivate(suma_pri)
    {
        #pragma omp for private(k)
        for (k=0; k<N; k++)
            suma_pri+=matriz[k][i];

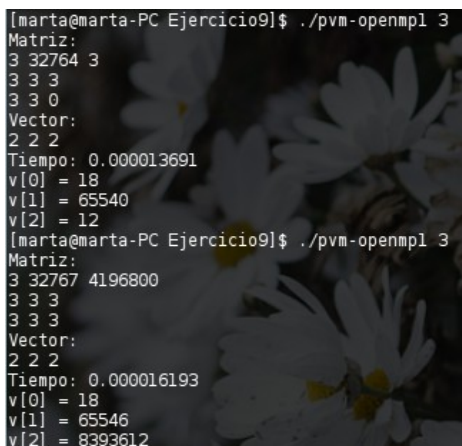
        #pragma omp atomic
        suma+=suma_pri;
    }

    vector_resultado[i] = suma * vector[i];
}
```

RESPUESTA: Al compilar la primera versión no me ha dado ningún fallo pues su implementación es sencilla, pero al inicializar las matrices había un pequeño problema que hacía que tomasen valores erróneos. Lo he solucionado de la siguiente manera:

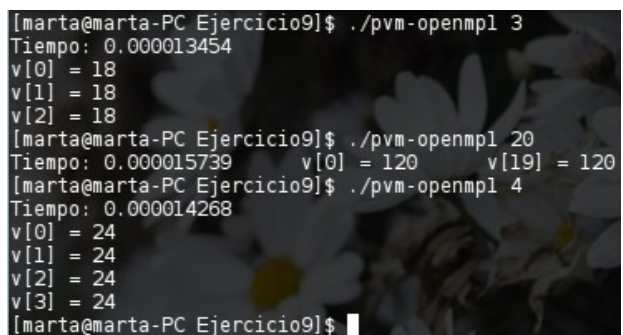
```
#pragma omp parallel for private(i,j)
```

Para hacer la segunda versión, me basé en lo que hice en el ejercicio 7 y me funcionó sin ningún problema.

CAPTURAS DE PANTALLA:


```
[marta@marta-PC Ejercicio9]$ ./pvm-openmpl 3
Matriz:
3 32764 3
3 3 3
3 3 0
Vector:
2 2 2
Tiempo: 0.000013691
v[0] = 18
v[1] = 65540
v[2] = 12
[marta@marta-PC Ejercicio9]$ ./pvm-openmpl 3
Matriz:
3 32767 4196800
3 3 3
3 3 3
Vector:
2 2 2
Tiempo: 0.000016193
v[0] = 18
v[1] = 65546
v[2] = 8393612
```

Captura_de_pantalla 1: Ejecución con resultados erróneos.



```
[marta@marta-PC Ejercicio9]$ ./pvm-openmpl 3
Tiempo: 0.000013454
v[0] = 18
v[1] = 18
v[2] = 18
[marta@marta-PC Ejercicio9]$ ./pvm-openmpl 20
Tiempo: 0.000015739 v[0] = 120 v[19] = 120
[marta@marta-PC Ejercicio9]$ ./pvm-openmpl 4
Tiempo: 0.000014268
v[0] = 24
v[1] = 24
v[2] = 24
v[3] = 24
[marta@marta-PC Ejercicio9]$
```

Captura_de_pantalla 2: Ejecución con resultados correctos

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).

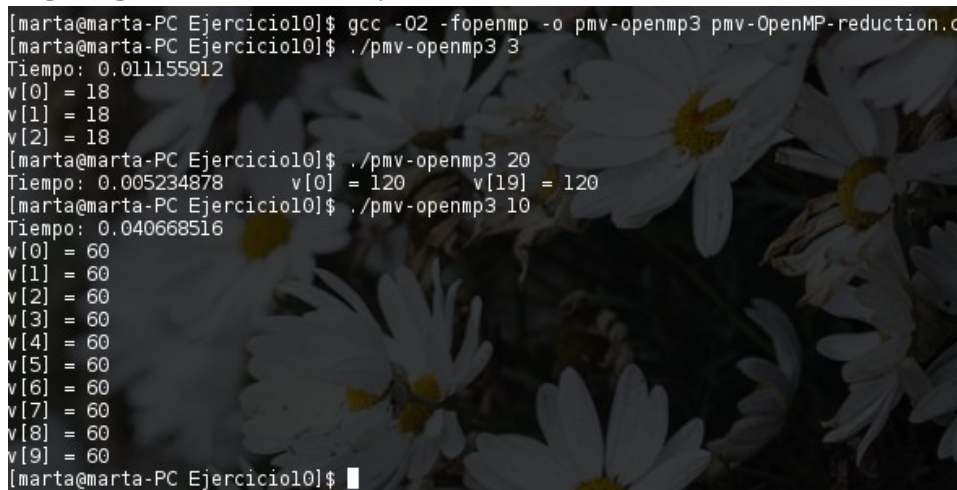
CÓDIGO FUENTE: pmv-OpenmMP-reduction.c

```
for (i=0; i<N; i++) {
    suma = 0;
    #pragma omp parallel for private(k), reduction(+:suma)
        for (k=0; k<N; k++)
            suma+=matriz[k][i];

    vector_resultado[i] = suma * vector[i];
}
```

RESPUESTA: Lo único que he hecho ha sido juntar las cláusulas parallel y for, y sustituir la directiva atomic por la cláusula reduction para hacer la suma.

CAPTURAS DE PANTALLA:



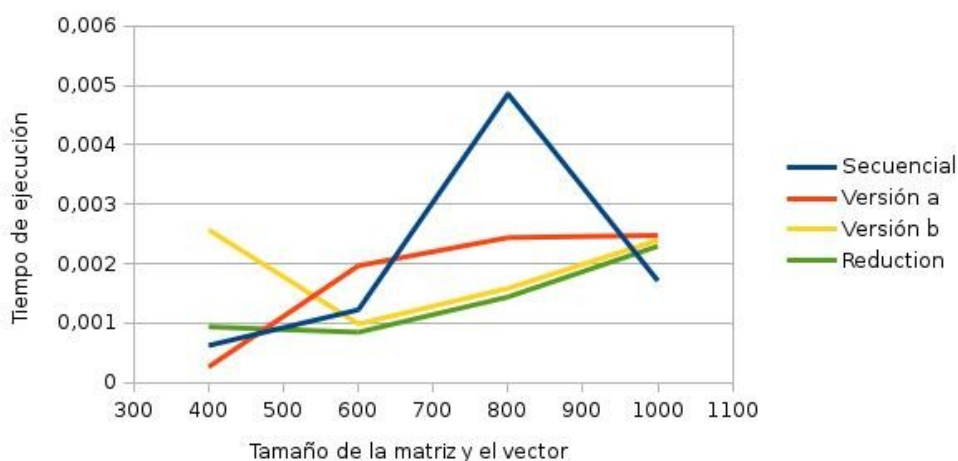
```
[marta@marta-PC Ejercicio10]$ gcc -O2 -fopenmp -o pmv-openmp3 pmv-OpenMP-reduction.c
[marta@marta-PC Ejercicio10]$ ./pmv-openmp3 3
Tiempo: 0.011155912
v[0] = 18
v[1] = 18
v[2] = 18
[marta@marta-PC Ejercicio10]$ ./pmv-openmp3 20
Tiempo: 0.005234878 v[0] = 120 v[19] = 120
[marta@marta-PC Ejercicio10]$ ./pmv-openmp3 10
Tiempo: 0.040668516
v[0] = 60
v[1] = 60
v[2] = 60
v[3] = 60
v[4] = 60
v[5] = 60
v[6] = 60
v[7] = 60
v[8] = 60
v[9] = 60
[marta@marta-PC Ejercicio10]$
```

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 el PC del aula de prácticas de los tres códigos implementados en los ejercicios anteriores para tres tamaños (N) distintos (consulte la Lección 6/Tema 2). Usar -O2 al compilar.

TABLA Y GRÁFICA (por ejemplo para 1-4 threads PC aula, y para 1-12 threads en atcgrid, tamaños-N-: 1.000, 10.000, 100.000):

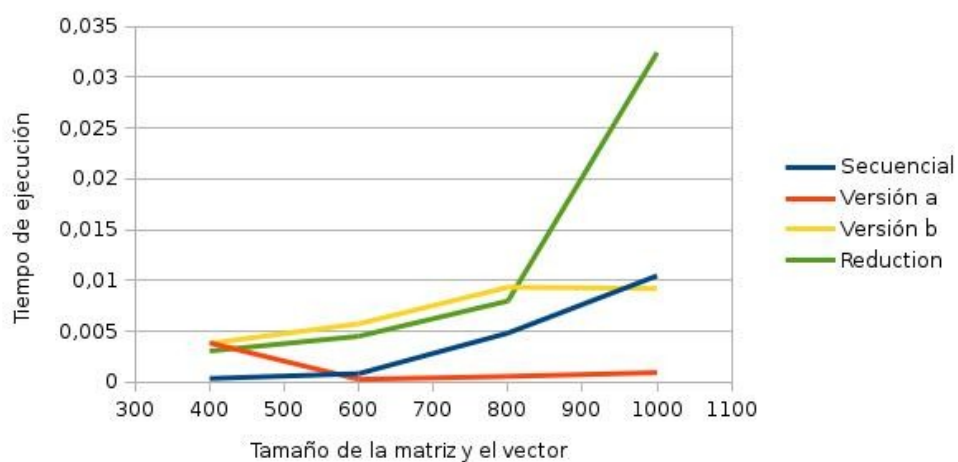
	Mi PC			
	Secuencial	Versión a	Versión b	reduction
400	0,000622356	0,000262047	0,002568450	0,000936696
600	0,001223354	0,001964308	0,000985140	0,000845663
800	0,004856650	0,002437130	0,001584115	0,001439357
1000	0,001710611	0,002474728	0,002393603	0,002293928

Resultados de ejecución en mi PC



	atcgrid			
	Secuencial	Versión a	Versión b	reduction
400	0,000336908	0,003848999	0,003798667	0,003048119
600	0,000818946	0,000250032	0,005730870	0,004495840
800	0,004812974	0,000543057	0,009324183	0,007965486
1000	0,010453837	0,000920078	0,009201628	0,032434959

Resultados obtenidos en atcgrid



COMENTARIOS SOBRE LOS RESULTADOS:

Los resultados parecen normales, pero no termino de estar de acuerdo con esos pequeños picos que hay en secuencial-800 y reduction-1000. Imagino que se deben a cualquier tipo de interrupción del sistema operativo, pero no dan un resultado representativo de la tendencia de la gráfica.

Obviando estos datos, podemos ver como la versión secuencial es la que más crece frente a las paralelas, y dentro de las paralelas, con pocos cores es mejor usar reduction pero cuando aumenta el número de cores la versión a del código es la más eficiente, como se aprecia en las gráficas.