

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): Juan Manuel Salcedo Serrano

Grupo de prácticas: B1

Fecha de entrega: 21 Abril

Fecha evaluación en clase: 21 Abril

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:

CÓDIGO FUENTE: `shared-clauseModificado.c`

```
#include <stdio.h>
#include <stdlib.h>
#ifdef _OPENMP
    #include <omp.h>
#endif
void main(){
    int i, n = 7;
    int a[n];

    for (i=0; i<n; i++) a[i] = i+1;

    #pragma omp parallel for default(none) shared(a,n)
    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:



```
+ BP2 gcc -O2 -o sharedmod shared-clauseModificado.c
+ BP2 ./sharedmod
Después de parallel for:
a[0] = 1
a[1] = 3
a[2] = 5
a[3] = 7
a[4] = 9
a[5] = 11
a[6] = 13
```

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:

Al inicializar `suma` fuera las distintas hebras no utilizarán este valor, sino que le darán uno indefinido.

CÓDIGO FUENTE: `private-clauseModificado.c`

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif

main() {
    int i, n = 7;
    int a[n], suma;

    for (i = 0; i < n; i++)
        a[i] = i;

    suma = 5;

#pragma omp parallel private(suma)
    {
        printf("\nValor de suma al inicio del parallel en hebra %d: %d\n", omp_get_thread_num(),
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("\nValor de suma fuera de parallel: %d", suma);

    printf("\n");
}
```

CAPTURAS DE PANTALLA:


```
➔ BP2 gcc -O2 -o priv2 private-clausemod.c
➔ BP2 ./priv2

Valor de suma al inicio del parallel en hebra 0: 5
thread 0 suma a[0] / thread 0 suma a[1] / thread 0 suma a[2] / thread 0 su
ma a[3] / thread 0 suma a[4] / thread 0 suma a[5] / thread 0 suma a[6] /
* thread 0 suma= 26
Valor de suma fuera de parallel: 26
```

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

RESPUESTA:

Al quitar la cláusula `private` todas las hebras comparten el mismo valor de `suma`, puesto que solo existe una copia, por tanto todas dan el mismo valor

CÓDIGO FUENTE: `private-clauseModificado3.c`

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif

main() {
    int i, n = 7;
    int a[n], suma;

    for (i = 0; i < n; i++)
        a[i] = i;

    #pragma omp parallel
    {
        suma = 5;

        printf("\nValor de suma al inicio del parallel en hebra %d: %d\n",
            omp_get_thread_num(), 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("\nValor de suma fuera de parallel: %d", suma);

    printf("\n");
}
```

CAPTURAS DE PANTALLA:

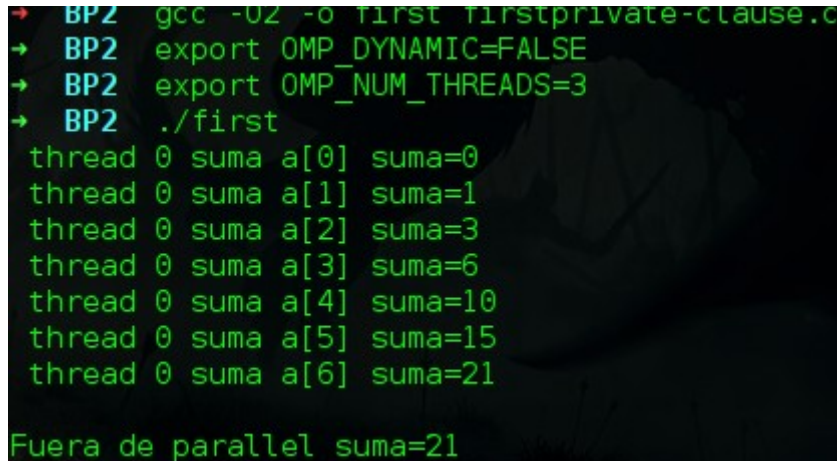
```
→ BP2 gcc -O2 -o priv3 private-clausemod3.c
→ BP2 ./priv3

Valor de suma al inicio del parallel en hebra 0: 5
thread 0 suma a[0] / thread 0 suma a[1] / thread 0 suma a[2] / thread 0 su
ma a[3] / thread 0 suma a[4] / thread 0 suma a[5] / thread 0 suma a[6] /
* thread 0 suma= 26
Valor de suma fuera de parallel: 26
```

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:

CAPTURAS DE PANTALLA:



```

BP2 gcc -O2 -fopenmp firstlastprivate.c
BP2 export OMP_DYNAMIC=FALSE
BP2 export OMP_NUM_THREADS=3
BP2 ./first
thread 0 suma a[0] suma=0
thread 0 suma a[1] suma=1
thread 0 suma a[2] suma=3
thread 0 suma a[3] suma=6
thread 0 suma a[4] suma=10
thread 0 suma a[5] suma=15
thread 0 suma a[6] suma=21
Fuera de parallel suma=21
    
```

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

RESPUESTA:

Al quitarle la cláusula `copyprivate(a)` a las hebras ya no se les asigna el mismo valor de `a` (leído por `scanf`), sino que al leerse solouna vez, el resto usan una variable sin inicializar, siendo su valor indeterminad.

CÓDIGO FUENTE: `copyprivate.c`Modificado.c

```

#include <stdio.h>
#include <omp.h>

main() {
    int n = 9, i, b[n];

    for (i = 0; i < n; i++) b[i] = -1;

    #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("Iteracion %d, Hebra %d, Valor de b[%d] = %d.\n", i,
omp_get_thread_num(), i, b[i]);
        }

    printf("Después de la región parallel:\n");
    for (i = 0; i < n; i++) printf("b[%d] = %d\t", i, b[i]);
    printf("\n");
}

```

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

RESPUESTA: Al iniciar suma fuera con valor 10, todos los resultados se incrementan en esa cantidad, ya que cuando dentro del parallel se realizan las sumas parciales no empiezan desde 0, sino desde 10.

CÓDIGO FUENTE: reduction-clauseModificado.c

```

#include <stdio.h>
#include <stdlib.h>
#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif

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:



```

BP2 gcc -O2 -o reduction reduction-clause.c
BP2 export OMP_NUM_THREADS=3
BP2 ./reduction 10
Tras 'parallel' suma=55
BP2 ./reduction 20
Tras 'parallel' suma=200
BP2 ./reduction 6
Tras 'parallel' suma=25

```

7. En el ejemplo `reduction-clause.c`, elimine `for` 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.

CÓDIGO FUENTE: `reduction-clauseModificado7.c`

```
#include <stdio.h>
#include <stdlib.h>
#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif

main(int argc, char **argv) {
    int i, n=20, a[n], suma=10;

    if(argc < 2) {
        fprintf(stderr, "Falta iteraciones\n");
        exit(-1);
    }

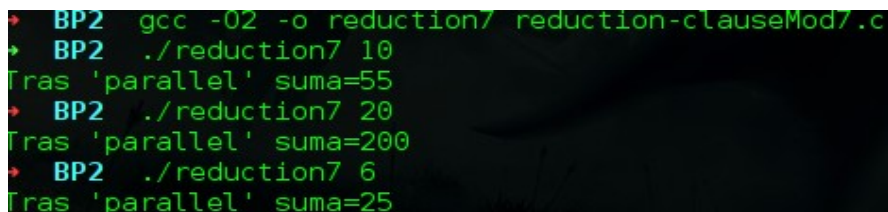
    n = atoi(argv[1]);
    if (n>20) {
        n=20;
        printf("n=%d", n);
    }

    for (i=0; i<n; i++) a[i] = i;

    #pragma omp parallel sections reduction(+:suma) private(i)
    {
        #pragma omp section
        for (i = 0; i < n/4; i++)
            suma += a[i];
        #pragma omp section
        for (i = n/4; i < n/2; i++)
            suma += a[i];
        #pragma omp section
        for (i = n/2; i < n/2+n/4; i++)
            suma += a[i];
        #pragma omp section
        for (i = n/2+n/4; i < n; i++)
            suma += a[i];
    }

    printf("Tras 'parallel' suma=%d\n", suma);
}
```

CAPTURAS DE PANTALLA:



```
BP2 gcc -O2 -o reduction7 reduction-clauseMod7.c
BP2 ./reduction7 10
Tras 'parallel' suma=55
BP2 ./reduction7 20
Tras 'parallel' suma=200
BP2 ./reduction7 6
Tras 'parallel' suma=25
```

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):

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

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

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {

    if (argc < 2) {
        fprintf(stderr, "\n Error, introducir dimension.
\n");
        exit(-1);
    }

    int i, k, N = atoi(argv[1]), VR[N], M[N][N], V[N];

    printf("\nValores vector: ");
    for (i = 0; i < N; i++) {
        VR[i] = 0;
        V[i] = i + 2;
        printf("[%d]", V[i]);
    }

    printf("\n\nValores matriz:");
    for (i = 0; i < N; i++) {
        printf("\n");
        for (k = 0; k < N; k++) {
            M[i][k] = i + k + 1;
            printf("[%d]", M[i][k]);
        }
    }

    for (i = 0; i < N; i++) {
        printf("\n%d: ", i);

        for (k = 0; k < N; k++) {
            VR[i] += M[i][k] * V[k];
            printf("->[%d]", VR[i]);
        }
    }
}
```



```

    }

    printf("\n\nValor primer componente resultado: %d", VR[0]);
    printf("\nValor ultimo componente resultado: %d", VR[N - 1]);

    printf("\n\n");
    return (0);
}

```

CAPTURAS DE PANTALLA:

```

→ BP2 ./producto 3

Valores vector: [2][3][4]

Valores matriz:
[1][2][3]
[2][3][4]
[3][4][5]
0: ->[2] ->[8] ->[20]
1: ->[4] ->[13] ->[29]
2: ->[6] ->[18] ->[38]

Valor primer componente resultado: 20
Valor ultimo componente resultado: 38

```

```

→ BP2 gcc -O2 -o producto producto.c
→ BP2 ./producto 10

Valores vector: [2][3][4][5][6][7][8][9][10][11]

Valores matriz:
[1][2][3][4][5][6][7][8][9][10]
[2][3][4][5][6][7][8][9][10][11]
[3][4][5][6][7][8][9][10][11][12]
[4][5][6][7][8][9][10][11][12][13]
[5][6][7][8][9][10][11][12][13][14]
[6][7][8][9][10][11][12][13][14][15]
[7][8][9][10][11][12][13][14][15][16]
[8][9][10][11][12][13][14][15][16][17]
[9][10][11][12][13][14][15][16][17][18]
[10][11][12][13][14][15][16][17][18][19]
0: ->[2] ->[8] ->[20] ->[40] ->[70] ->[112] ->[168] ->[240] ->[330] ->[440]
1: ->[4] ->[13] ->[29] ->[54] ->[90] ->[139] ->[203] ->[284] ->[384] ->[505]
2: ->[6] ->[18] ->[38] ->[68] ->[110] ->[166] ->[238] ->[328] ->[438] ->[570]
3: ->[8] ->[23] ->[47] ->[82] ->[130] ->[193] ->[273] ->[372] ->[492] ->[635]
4: ->[10] ->[28] ->[56] ->[96] ->[150] ->[220] ->[308] ->[416] ->[546] ->[700]
5: ->[12] ->[33] ->[65] ->[110] ->[170] ->[247] ->[343] ->[460] ->[600] ->[765]
6: ->[14] ->[38] ->[74] ->[124] ->[190] ->[274] ->[378] ->[504] ->[654] ->[830]
7: ->[16] ->[43] ->[83] ->[138] ->[210] ->[301] ->[413] ->[548] ->[708] ->[895]
8: ->[18] ->[48] ->[92] ->[152] ->[230] ->[328] ->[448] ->[592] ->[762] ->[960]
9: ->[20] ->[53] ->[101] ->[166] ->[250] ->[355] ->[483] ->[636] ->[816] ->[1025]

Valor primer componente resultado: 440
Valor ultimo componente resultado: 1025

```


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

```
#include <stdio.h>
#include <stdlib.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif

int main(int argc, char **argv) {

    if (argc < 2) {
        fprintf(stderr, "\n[ERROR] - Debe introducir introducir dimension. \n");
        exit(-1);
    }

    int i, k, N = atoi(argv[1]), VR[N], M[N][N], V[N];

    printf("\nIniciando vector...");
    #pragma omp parallel for
    {
        for (i = 0; i < N; i++) {
            printf("\nHebra %d: VR[%d], V[%d]", omp_get_thread_num(), i, i);
            VR[i] = 0;
            V[i] = i + 2;
        }
    }

    printf("\n\nValores vector: ");
```

```

    for (i = 0; i < N; i++) printf("[%d]", V[i]);

    printf("\n\nInicializando matriz...");
}
#pragma omp parallel for
{
    for (i = 0; i < N; i++) {
        printf("\nHebra %d, fila %d: ", omp_get_thread_num(), i);

        for (k = 0; k < N; k++) {
            M[i][k] = i + k + 1;
            printf("[%d]", M[i][k]);
        }
    }
    printf("\n\nCalculando producto...");
#pragma omp parallel for
{
    for (i = 0; i < N; i++) {
        printf("\nHebra %d calcula componente %d: ", omp_get_thread_num(), i);

        for (k = 0; k < N; k++) {
            VR[i] += M[i][k] * V[k];
            printf("->[%d]", VR[i]);
        }
    }

    printf("\n\nValor primera componente resultado: %d", VR[0]);
    printf("\nValor ultima componente resultado: %d", VR[N - 1]);

    printf("\n\n");
    return (0);
}
}

```

CÓDIGO FUENTE: pmv-OpenMP-b.c

```

#include <stdio.h>
#include <stdlib.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif

int main(int argc, char **argv) {

    if (argc < 2) {
        fprintf(stderr, "\n[ERROR] - Debe introducir dimension. \n");
        exit(-1);
    }

    int i, k, N = atoi(argv[1]), VR[N], M[N][N], V[N];

```

```

    printf("\nIniciando vector...");
#pragma omp parallel for
    for (i = 0; i < N; i++) {
        printf("\nHebra %d: VR[%d], V[%d]", omp_get_thread_num(), i, i);
        VR[i] = 0;
        V[i] = i + 2;
    }

    printf("\n\nValores vector: ");
    for (i = 0; i < N; i++) printf("[%d]", V[i]);

    printf("\n\nIniciando matriz...");
    for (i = 0; i < N; i++) {
#pragma omp parallel for lastprivate(k)
        for (k = 0; k < N; k++) {
            printf("\nHebra %d, componente [%d][%d]", omp_get_thread_num(), i, k);
            M[i][k] = i + k + 1;
        }
    }

    printf("\n\nValores matriz: ");
    for (i = 0; i < N; i++) {
        printf("\n");
        for (k = 0; k < N; k++) {
            M[i][k] = i + k + 1;
            printf("[%d]", M[i][k]);
        }
    }

    printf("\n\nCalculando producto...");
    for (i = 0; i < N; i++) {
#pragma omp parallel for lastprivate(k)
        for (k = 0; k < N; k++) {
            VR[i] += M[i][k] * V[k];

            printf("\nHebra %d, componente %d, operacion %d: ->[%d]",
omp_get_thread_num(), i, k, VR[i]);
        }
    }

    printf("\n\nValor primera componente resultado: %d", VR[0]);
    printf("\n\nValor ultima componente resultado: %d", VR[N - 1]);

    printf("\n\n");
    return (0);
}

```

CAPTURAS DE PANTALLA:

```
→ BP2 gcc -O2 -o pmva pmv-a.c
→ BP2 ./pmva 3

Inicializando vector...
Hebra 0: VR[0], V[0]
Hebra 0: VR[1], V[1]
Hebra 0: VR[2], V[2]

Valores vector: [2][3][4]

Inicializando matriz...
Hebra 0, fila 0: [1][2][3]
Hebra 0, fila 1: [2][3][4]
Hebra 0, fila 2: [3][4][5]

Calculando producto...
Hebra 0 calcula componente 0: ->[2]->[8]->[20]
Hebra 0 calcula componente 1: ->[4]->[13]->[29]
Hebra 0 calcula componente 2: ->[6]->[18]->[38]

Valor primera componente resultado: 20
Valor ultima componente resultado: 38
```

```
→ BP2 gcc -O2 -o pmvb pmv-b.c
→ BP2 ./pmvb 3

Inicializando vector...
Hebra 0: VR[0], V[0]
Hebra 0: VR[1], V[1]
Hebra 0: VR[2], V[2]

Inicializando matriz...
Hebra 0, componente[0][0]
Hebra 0, componente[0][1]
Hebra 0, componente[0][2]
Hebra 0, componente[1][0]
Hebra 0, componente[1][1]
Hebra 0, componente[1][2]
Hebra 0, componente[2][0]
Hebra 0, componente[2][1]
Hebra 0, componente[2][2]

Calculando producto...
Hebra 0, componente 0, operacion 0: ->[2]
Hebra 0, componente 0, operacion 1: ->[8]
Hebra 0, componente 0, operacion 2: ->[20]
Hebra 0, componente 1, operacion 0: ->[4]
Hebra 0, componente 1, operacion 1: ->[13]
Hebra 0, componente 1, operacion 2: ->[29]
Hebra 0, componente 2, operacion 0: ->[6]
Hebra 0, componente 2, operacion 1: ->[18]
Hebra 0, componente 2, operacion 2: ->[38]

Valor primera componente resultado: 20
Valor ultima componente resultado: 38
```

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-OpenMP-reduction.c

```
#include <stdio.h>
#include <stdlib.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif

int main(int argc, char **argv) {

    if (argc < 2) {
        fprintf(stderr, "\n[ERROR] - Debe introducir introducir dimension. \n");
        exit(-1);
    }

    int i, k, N = atoi(argv[1]), VR[N], M[N][N], V[N], producto = 0;

    printf("\nIniciando vector...");

    #pragma omp parallel for
    for (i = 0; i < N; i++) {
        printf("\nHebra %d: VR[%d], V[%d]", omp_get_thread_num(), i, i);
        VR[i] = 0;
        V[i] = i + 2;
    }

    printf("\n\nValores vector: ");
    for (i = 0; i < N; i++) printf("[%d]", V[i]);

    printf("\n\nIniciando matriz...");
    #pragma omp parallel for
    for (i = 0; i < N; i++) {
        printf("\nHebra %d, fila %d: ", omp_get_thread_num(), i);

        for (k = 0; k < N; k++) {
            M[i][k] = i + k + 1;
            printf("[%d]", M[i][k]);
        }
    }

    printf("\n\nCalculando producto...");
    #pragma omp parallel for reduction(+:producto)
    for (i = 0; i < N; i++) {
        producto = 0;
        printf("\nHebra %d calcula componente %d: ", omp_get_thread_num(), i);
```

```

    for (k = 0; k < N; k++) {
        producto += M[i][k] * V[k];
        printf("->[%d]", producto);
    }

    VR[i] = producto;
}

printf("\n\nValor primera componente resultado: %d", VR[0]);
printf("\nValor ultima componente resultado: %d", VR[N - 1]);

printf("\n\n");

return (0);
}

```

RESPUESTA:

CAPTURAS DE PANTALLA:



```

→ BP2 gcc -O2 -o pmvreduction pmv-reduction.c
→ BP2 ./pmvreduction 3

Inicializando vector...
Hebra 0: VR[0], V[0]
Hebra 0: VR[1], V[1]
Hebra 0: VR[2], V[2]

Valores vector: [2][3][4]

Inicializando matriz...
Hebra 0, fila 0: [1][2][3]
Hebra 0, fila 1: [2][3][4]
Hebra 0, fila 2: [3][4][5]

Calculando producto...
Hebra 0 calcula componente 0: ->[2]->[8]->[20]
Hebra 0 calcula componente 1: ->[4]->[13]->[29]
Hebra 0 calcula componente 2: ->[6]->[18]->[38]

Valor primera componente resultado: 20
Valor ultima componente resultado: 38

```