2º curso / 2º cuatr.

Grado Ing. Inform.

Doble Grado Ing. Inform. y Mat.

Arquitectura de Computadores (AC)

Cuaderno de prácticas.
Bloque Práctico 1. Programación paralela I: Directivas OpenMP

Estudiante (nombre y apellidos): Antonio Martín Ruiz
Grupo de prácticas:2
Fecha de entrega: 30/03/2017
Fecha evaluación en clase:31/03/2017

Ejercicios basados en los ejemplos del seminario práctico

. Usar la directiva `parallel` combinada con directivas de trabajo compartido en los ejemplos `bucle-for.c` y `sections.c` del seminario. Incorporar el código fuente resultante al cuaderno de prácticas.

RESPUESTA: código fuente `bucle-forModificado.c`

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


int main (int argc, char** argv) {
  int i,n = 9 ;


  if (argc<2) {
    fprintf(stderr, "\n[ERROR] - Falta nº de iteraciones \n");
    exit(-1);
  }
  n = atoi(argv[1]);


  #pragma omp parallel for


    for (i = 0; i < n; i++)
      printf("thread %d ejecuta la iteración %d del bucle\n",
omp_get_thread_num(), i);


  return (0);
}
```

RESPUESTA: código fuente `sectionsModificado.c`

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


void funcA() {
  printf("En funcA: esta sección la ejecuta el thread %d\n",
omp_get_thread_num());
}


void funcB() {
  printf("En funcB: esta sección la ejecuta el thread %d\n",
```

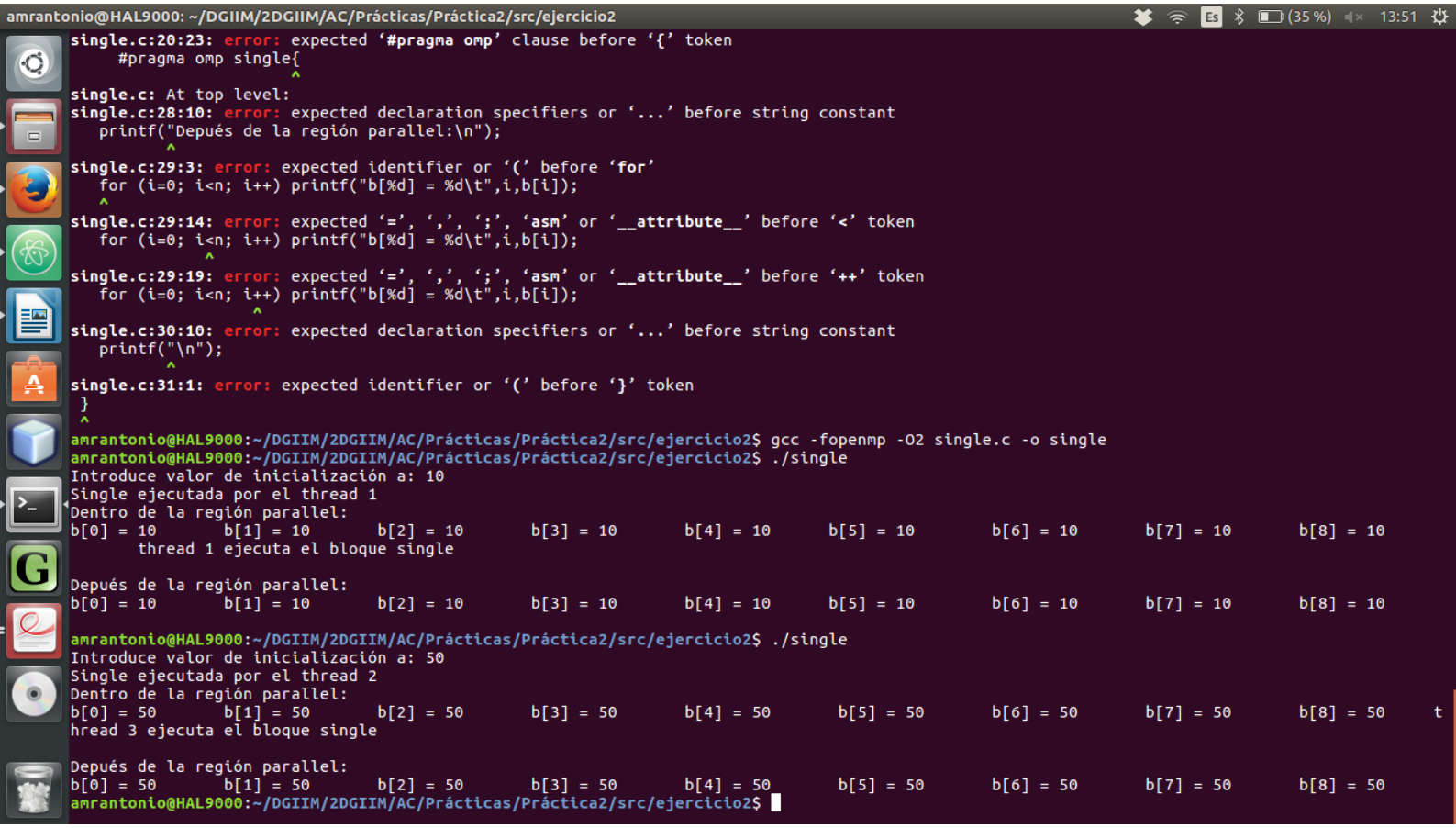
```
omp_get_thread_num());
}


int main() {
  #pragma omp parallel sections
  {
      #pragma omp section
      (void) funcA();
      #pragma omp section
      (void) funcB();
  }
}
```

. Imprimir los resultados del programa `single.c` usando una directiva `single` dentro de la construcción `parallel` en lugar de imprimirlos fuera de la región `parallel`. Añadir lo necesario, dentro de la nueva directiva `single` incorporada, para que se imprima el identificador del thread que ejecuta el bloque estructurado de la directiva `single`. Incorpore en su cuaderno de trabajo el código fuente y volcados de pantalla con los resultados de ejecución obtenidos.

RESPUESTA: código fuente `singleModificado.c`

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


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


  for (i=0; i<n; i++) b[i] = -1;
  #pragma omp parallel
  {
    #pragma omp single
    { printf("Introduce valor de inicialización a: ");
      scanf("%d", &a );
      printf("Single ejecutada por el thread %d\n", omp_get_thread_num());
    }


    #pragma omp for
      for (i=0; i<n; i++)
        b[i] = a;


    #pragma omp single
    {
      printf("Dentro de la región parallel:\n");
      for (i=0; i<n; i++) printf("b[%d] = %d\t",i,b[i]);
      printf("thread %d ejecuta el bloque single\n", omp_get_thread_num());
      printf("\n");
    }


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

CAPTURAS DE PANTALLA:



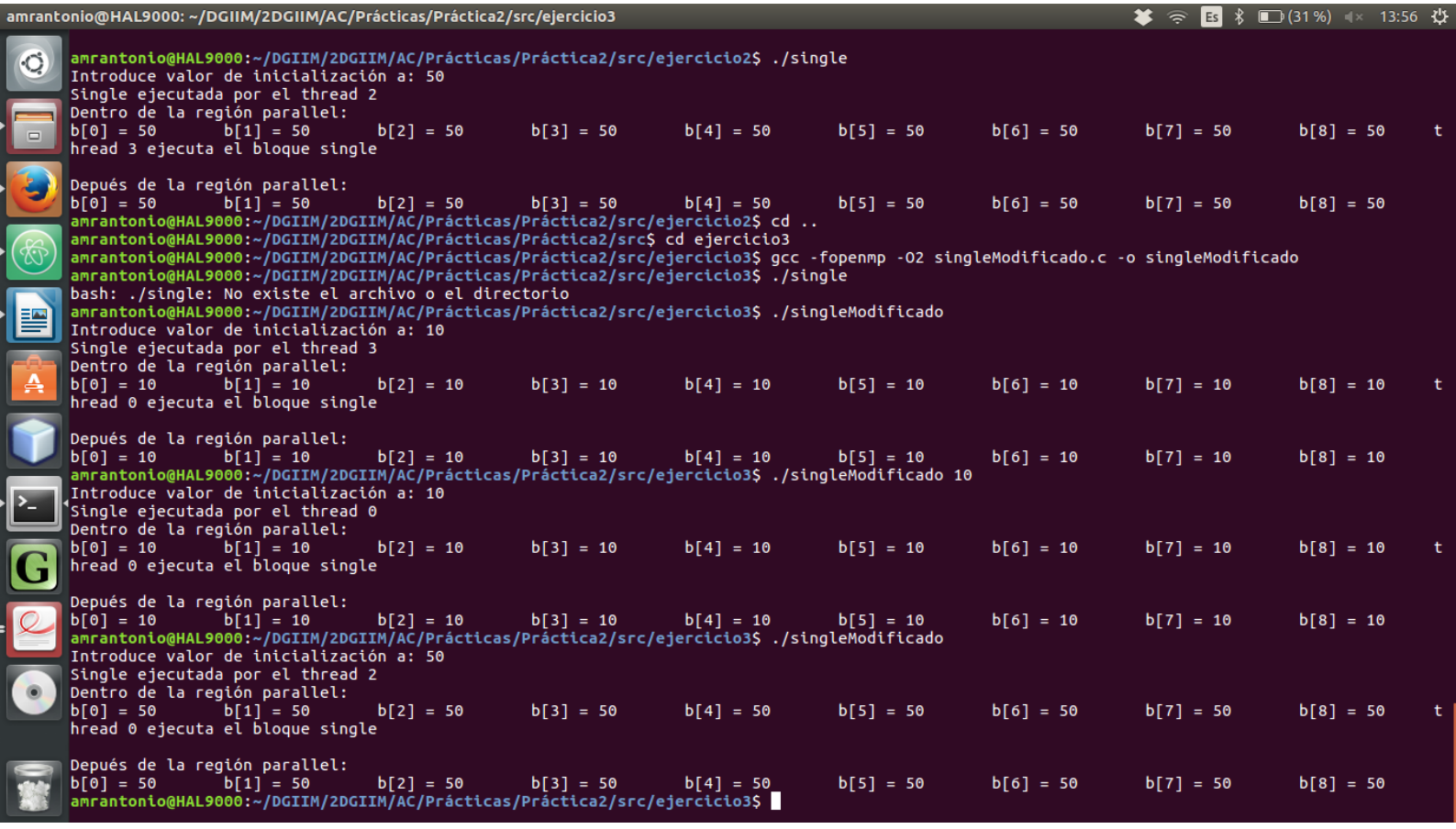
. Imprimir los resultados del programa `single.c` usando una directiva `master` dentro de la construcción `parallel` en lugar de imprimirlos fuera de la región `parallel`. Añadir lo necesario, dentro de la nueva directiva `master` incorporada, para que se imprima el identificador del thread que ejecuta el bloque estructurado de la directiva `master`. Incorpore en su cuaderno el código fuente y volcados de pantalla con los resultados de ejecución obtenidos. ¿Qué diferencia observa con respecto a los resultados de ejecución del ejercicio anterior?

RESPUESTA: código fuente `singleModificado2.c`

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


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


  for (i=0; i<n; i++) b[i] = -1;
  #pragma omp parallel
  {
    #pragma omp single
    { printf("Introduce valor de inicialización a: ");
      scanf("%d", &a );
      printf("Single ejecutada por el thread %d\n", omp_get_thread_num());
    }
```

```
    #pragma omp for
      for (i=0; i<n; i++)
        b[i] = a;


    #pragma omp master
    {
      printf("Dentro de la región parallel:\n");
      for (i=0; i<n; i++) printf("b[%d] = %d\t",i,b[i]);
      printf("thread %d ejecuta el bloque single\n", omp_get_thread_num());
      printf("\n");
    }


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

CAPTURAS DE PANTALLA:



RESPUESTA A LA PREGUNTA: La diferencia es que en el primer caso cualquier hebra puede ejecutar el bloque, mientras que en el segundo, solo la hebra maestra puede hacerlo. Esto se manifiesta en que la única hebra que ejecuta el bloque con master es la 0.

. ¿Por qué si se elimina directiva `barrier` en el ejemplo `master.c` la suma que se calcula e imprime no siempre es correcta? Responda razonadamente.

RESPUESTA: Porque si el thread master acaba antes que algún otro, imprimirá el resultado sin esperar a que los que acaben después sumen su parte a la suma total.

Resto de ejercicios

. El programa secuencial C del Listado 1 calcula la suma de dos vectores (v3 = v1 + v2; v3(i) = v1(i) + v2(i), i=0,…N-1). Generar el ejecutable del programa del Listado 1 para **vectores globales**. Usar `time` (Lección 3/ Tema 1) en la línea de comandos para obtener, en el PC local, el tiempo de ejecución (*elapsed time*) y el tiempo de CPU del usuario y del sistema generado. Obtenga los tiempos para vectores con 10000000 componentes. ¿La suma de los tiempos de CPU del usuario y del sistema es mayor o igual que el tiempo real (*elapsed*)? Justifique la respuesta.

CAPTURAS DE PANTALLA:

```
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/ejercicio5$ gcc -O2 -lrt SumaVectores.c -o SumaVectores
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/ejercicio5$ time ./SumaVectores 10000000
Tiempo(seg.):0.036504016        / Tamaño Vectores:10000000      / V1[0]+V2[0]=V3[0](1000000.000000+1000000.000000=2000000.000000) / /V1[9999999
]+V2[9999999]=V3[9999999](1999999.900000+0.100000=2000000.000000) /

real    0m0.086s
user    0m0.060s
sys     0m0.024s
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/ejercicio5$
```

Ka suma de usuario y sistema es prácticamente igual a la real. Esto es debido a que el programa se ejecuta secuencialmente.

. Generar el código ensamblador a partir del programa secuencial C del Listado 1 para **vectores globales** (para generar el código ensamblador tiene que compilar usando `-S` en lugar de `-o`). Utilice el fichero con el código fuente ensamblador generado y el fichero ejecutable generado en el ejercicio 5 para obtener para atcgrid los MIPS (*Millions of Instructions Per Second*) y los MFLOPS (*Millions of FLOating-point Per Second*) del código que obtiene la suma de vectores (código entre las funciones `clock_gettime()`); el cálculo se debe hacer para 10 y 10000000 componentes en los vectores (consulte la Lección 3/Tema1 AC). Incorpore **el código ensamblador de la parte de la suma de vectores** en el cuaderno.

CAPTURAS DE PANTALLA:

```
[E1estudiante13@atcgrid ~]$ echo './SumaVectores 10' | qsub -q ac
51833.atcgrid
[E1estudiante13@atcgrid ~]$ cat STDIN.e51833
[E1estudiante13@atcgrid ~]$ cat STDIN.o51833
Tiempo(seg.):0.000000182        / Tamaño Vectores:10    / V1[0]+V2[0]=V3[0](1.000000+1.000000=2.000000) / /V1[9]+V2[9]=V3[9](1.900000+0.100000=
2.000000) /
[E1estudiante13@atcgrid ~]$ echo './SumaVectores 10000000' | qsub -q ac
51840.atcgrid
[E1estudiante13@atcgrid ~]$ cat STDIN.e51840
[E1estudiante13@atcgrid ~]$ cat STDIN.o51840
Tiempo(seg.):0.047298628        / Tamaño Vectores:10000000      / V1[0]+V2[0]=V3[0](1000000.000000+1000000.000000=2000000.000000) / /V1[9999999
]+V2[9999999]=V3[9999999](1999999.900000+0.100000=2000000.000000) /
```

RESPUESTA: cálculo de los MIPS y los MFLOPS

Para 10 componentes

MIPS:

(6 instrucciones/elemento del vector * 10 elementos del vector)/(0.000000182s *10^6) = 329.67032967 MIPS

MFLOPS:

(3 flops/elemento del vector * 10 elementos del vector)/(0.000000182s *10^6) = 164.835164835

Para 10000000 componentes

MIPS:

(6 instrucciones/elemento del vector * 10000000 elementos del vector)/(0.047298628s*10^6) = 1268.5357385 MIPS

MFLOPS:

(3 flops/elemento del vector * 10 elementos del vector)/(0.047298628s*10^6) = 634.26786925 MFLOPS

RESPUESTA:

código ensamblador generado de la parte de la suma de vectores

```
.L9:
                movsd           0(%rbp,%rax,8), %xmm0
                addsd           (%r12,%rax,8), %xmm0
                movsd           %xmm0, (%r15,%rax,8)
                addq            $1, %rax
                cmpl            %eax, %r14d
                ja              .L9
```

. Implementar un programa en C con OpenMP, a partir del código del Listado 1, que calcule en paralelo la suma de dos vectores (v3 = v1 + v2; v3(i)=v1(i)+v2(i), i=0,…N-1) usando las directivas `parallel` y `for`. Se debe paralelizar también las tareas asociadas a la inicialización de los vectores. Como en el código del Listado 1 se debe obtener el tiempo (*elapsed time*) que supone el cálculo de la suma. Para obtener este tiempo usar la función `omp_get_wtime()`**,** que proporciona el estándar OpenMP, en lugar de `clock_gettime()`. NOTAS: (1) el número de componentes N de los vectores debe ser un argumento de entrada al programa; (2) se deben inicializar los vectores antes del cálculo; (3) se debe asegurar que el programa calcula la suma correctamente imprimiendo todos los componentes del vector resultante, v3, para varios 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 suma los vectores y, al menos, el primer y último componente de v1, v2 y v3 (esto último evita que las optimizaciones del compilador eliminen el código de la suma).

RESPUESTA: código fuente implementado

```
#include <stdlib.h> // biblioteca con funciones atoi(), malloc() y free()
#include <stdio.h> // biblioteca donde se encuentra la función printf()
#include <time.h> // biblioteca donde se encuentra la función clock_gettime()
//#define PRINTF_ALL // comentar para quitar el printf ...
// que imprime todos los componentes
#ifdef _OPENMP
  #include <omp.h>
#else
  #define omp_get_thread_num() 0
  #define omp_get_num_threads() 1
#endif
//Sólo puede estar definida una de las tres constantes VECTOR_ (sólo uno de
los ...
//tres defines siguientes puede estar descomentado):
//#define VECTOR_LOCAL // descomentar para que los vectores sean variables ...
// locales (si se supera el tamaño de la pila se ...
// generará el error "Violación de Segmento")
#define VECTOR_GLOBAL// descomentar para que los vectores sean variables ...
// globales (su longitud no estará limitada por el ...
// tamaño de la pila del programa)
//#define VECTOR_DYNAMIC // descomentar para que los vectores sean
variables ...
// dinámicas (memoria reutilizable durante la ejecución)
```

```
#ifdef VECTOR_GLOBAL
#define MAX 33554432 //=2^25
double v1[MAX], v2[MAX], v3[MAX];
#endif
int main(int argc, char** argv){
int i;
double cgt1,cgt2; double ncgt; //para tiempo de ejecución
//Leer argumento de entrada (no de componentes del vector)
if (argc<2){
printf("Faltan no componentes del vector\n");
exit(-1);
}
unsigned int N = atoi(argv[1]); // Máximo N =2^32-1=4294967295
(sizeof(unsigned int) = 4 B)
#ifdef VECTOR_LOCAL
double v1[N], v2[N], v3[N];
// Tamaño variable local en tiempo de ejecución ...
// disponible en C a partir de actualización C99
#endif
#ifdef VECTOR_GLOBAL
if (N>MAX) N=MAX;
#endif
#ifdef VECTOR_DYNAMIC
double *v1, *v2, *v3;
v1 = (double*) malloc(N*sizeof(double));// malloc necesita el tamaño en bytes
v2 = (double*) malloc(N*sizeof(double)); //si no hay espacio suficiente malloc
devuelve NULL
v3 = (double*) malloc(N*sizeof(double));
if ( (v1==NULL) || (v2==NULL) || (v3==NULL) ){
printf("Error en la reserva de espacio para los vectores\n");
exit(-2);
}
#endif
//Inicializar vectores
#pragma omp parallel
{
  #pragma omp for
  for(i=0; i<N; i++){
    v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1; //los valores dependen de N
  }
  #pragma omp single
  {
    cgt1 = omp_get_wtime();
  }
//Calcular suma de vectores
  #pragma omp for
  for(i=0; i<N; i++)
    v3[i] = v1[i] + v2[i];
  #pragma omp single
  {
    cgt2 = omp_get_wtime();
  }
}


ncgt=cgt2-cgt1;
//Imprimir resultado de la suma y el tiempo de ejecución
#ifdef PRINTF_ALL
printf("Tiempo(seg.):%11.9f\t / Tamaño Vectores:%u\n",ncgt,N);
for(i=0; i<N; i++)
printf("/ V1[%d]+V2[%d]=V3[%d](%8.6f+%8.6f=%8.6f)
/\n",i,i,i,v1[i],v2[i],v3[i]);
#else
printf("Tiempo(seg.):%11.9f\t / Tamaño Vectores:%u\t/ V1[0]+V2[0]=V3[0](%8.6f+
%8.6f=%8.6f) / /V1[%d]+V2[%d]=V3[%d](%8.6f+%8.6f=%8.6f) /\n",
ncgt,N,v1[0],v2[0],v3[0],N-1,N-1,N-1,v1[N-1],v2[N-1],v3[N-1]);
#endif
#ifdef VECTOR_DYNAMIC
```

```
free(v1); // libera el espacio reservado para v1
free(v2); // libera el espacio reservado para v2
free(v3); // libera el espacio reservado para v3
#endif
return 0;
}
```

(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)

CAPTURAS DE PANTALLA (compilación y ejecución para N=8 y N=11):

```
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/ejercicio7$ gcc -O2 -lrt -fopenmp SumaVectores.c -o SumaVectores
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/ejercicio7$ ./SumaVectores 8
Tiempo(seg.):0.001970871        / Tamaño Vectores:8    / V1[0]+V2[0]=V3[0](0.800000+0.800000=1.600000) / /V1[7]+V2[7]=V3[7](1.500000+0.100000=
1.600000) /
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/ejercicio7$ ./SumaVectores 11
Tiempo(seg.):0.000003783        / Tamaño Vectores:11   / V1[0]+V2[0]=V3[0](1.100000+1.100000=2.200000) / /V1[10]+V2[10]=V3[10](2.100000+0.1000
00=2.200000) /
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/ejercicio7$
```

. Implementar un programa en C con OpenMP, a partir del código del Listado 1, que calcule en paralelo la suma de dos vectores usando las `parallel` y `sections/section` (se debe aprovechar el paralelismo de datos usando estas directivas en lugar de la directiva `for`); es decir, hay que repartir el trabajo (tareas) entre varios threads usando `sections/section`. Se debe paralelizar también las tareas asociadas a la inicialización de los vectores. Para obtener este tiempo usar la función `omp_get_wtime()` en lugar de `clock_gettime()`. NOTAS: (1) el número de componentes N de los vectores debe ser un argumento de entrada al programa; (2) se deben inicializar los vectores 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 suma los vectores y, al menos, el primer y último componente de v1, v2 y v3 (esto último evita que las optimizaciones del compilador eliminen el código de la suma).

RESPUESTA: código fuente implementado

```
/* SumaVectoresC.c
Suma de dos vectores: v3 = v1 + v2
Para compilar usar (-lrt: real time library):
gcc -O2 SumaVectores.c -o SumaVectores –lrt
gcc -O2 –S SumaVectores.c –lrt
//para generar el código ensamblador
Para ejecutar use: SumaVectoresC longitud
*/
#include <stdlib.h> // biblioteca con funciones atoi(), malloc() y free()
#include <stdio.h> // biblioteca donde se encuentra la función printf()
#include <time.h> // biblioteca donde se encuentra la función clock_gettime()
//#define PRINTF_ALL // comentar para quitar el printf ...
// que imprime todos los componentes
#ifdef _OPENMP
  #include <omp.h>
#else
  #define omp_get_thread_num() 0
  #define omp_get_num_threads() 1
#endif
//Sólo puede estar definida una de las tres constantes VECTOR_ (sólo uno de
los ...
//tres defines siguientes puede estar descomentado):
//#define VECTOR_LOCAL // descomentar para que los vectores sean variables ...
// locales (si se supera el tamaño de la pila se ...
// generará el error "Violación de Segmento")
#define VECTOR_GLOBAL// descomentar para que los vectores sean variables ...
// globales (su longitud no estará limitada por el ...
```

```
// tamaño de la pila del programa)
//#define VECTOR_DYNAMIC // descomentar para que los vectores sean
variables ...
// dinámicas (memoria reutilizable durante la ejecución)
#ifdef VECTOR_GLOBAL
#define MAX 33554432 //=2^25
double v1[MAX], v2[MAX], v3[MAX];
#endif
int main(int argc, char** argv){
int i;
double cgt1,cgt2; double ncgt; //para tiempo de ejecución
//Leer argumento de entrada (no de componentes del vector)
if (argc<2){
printf("Faltan no componentes del vector\n");
exit(-1);
}
unsigned int N = atoi(argv[1]); // Máximo N =2^32-1=4294967295
(sizeof(unsigned int) = 4 B)
#ifdef VECTOR_LOCAL
double v1[N], v2[N], v3[N];
// Tamaño variable local en tiempo de ejecución ...
// disponible en C a partir de actualización C99
#endif
#ifdef VECTOR_GLOBAL
if (N>MAX) N=MAX;
#endif
#ifdef VECTOR_DYNAMIC
double *v1, *v2, *v3;
v1 = (double*) malloc(N*sizeof(double));// malloc necesita el tamaño en bytes
v2 = (double*) malloc(N*sizeof(double)); //si no hay espacio suficiente malloc
devuelve NULL
v3 = (double*) malloc(N*sizeof(double));
if ( (v1==NULL) || (v2==NULL) || (v3==NULL) ){
printf("Error en la reserva de espacio para los vectores\n");
exit(-2);
}
#endif
//Inicializar vectores
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
      for(i=0; i<N/4; i++){
        v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1; //los valores dependen de N
      }


    #pragma omp section
      for(i=N/4; i<N/2; i++){
        v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1; //los valores dependen de N
      }


    #pragma omp section
      for(i=N/2; i<3*N/4; i++){
        v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1; //los valores dependen de N
      }


    #pragma omp section
      for(i=3*N/4; i<N; i++){
        v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1; //los valores dependen de N
      }
    }


  #pragma omp single
```

```
  {
    cgt1 = omp_get_wtime();
  }
//Calcular suma de vectores
#pragma omp sections
{
  #pragma omp section
  for(i=0; i<N/4; i++)
    v3[i] = v1[i] + v2[i];
  #pragma omp section
  for(i=N/4; i<N/2; i++)
    v3[i] = v1[i] + v2[i];
  #pragma omp section
  for(i=N/2; i<3*N/4; i++)
    v3[i] = v1[i] + v2[i];
  #pragma omp section
  for(i=3*N/4; i<N; i++)
    v3[i] = v1[i] + v2[i];
}
  #pragma omp single
  {
    cgt2 = omp_get_wtime();
  }
}


ncgt=cgt2-cgt1;
//Imprimir resultado de la suma y el tiempo de ejecución
#ifdef PRINTF_ALL
printf("Tiempo(seg.):%11.9f\t / Tamaño Vectores:%u\n",ncgt,N);
for(i=0; i<N; i++)
printf("/ V1[%d]+V2[%d]=V3[%d](%8.6f+%8.6f=%8.6f)
/\n",i,i,i,v1[i],v2[i],v3[i]);
#else
printf("Tiempo(seg.):%11.9f\t / Tamaño Vectores:%u\t/ V1[0]+V2[0]=V3[0](%8.6f+
%8.6f=%8.6f) / /V1[%d]+V2[%d]=V3[%d](%8.6f+%8.6f=%8.6f) /\n",
ncgt,N,v1[0],v2[0],v3[0],N-1,N-1,N-1,v1[N-1],v2[N-1],v3[N-1]);
#endif
#ifdef VECTOR_DYNAMIC
free(v1); // libera el espacio reservado para v1
free(v2); // libera el espacio reservado para v2
free(v3); // libera el espacio reservado para v3
#endif
return 0;
}
```

(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)

CAPTURAS DE PANTALLA (compilación y ejecución para N=8 y N=11):

```
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/ejercicio8$ gcc -O2 -lrt -fopenmp SumaVectores.c -o SumaVectores
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/ejercicio8$ ./SumaVectores 8
Tiempo(seg.):0.000001580        / Tamaño Vectores:8    / V1[0]+V2[0]=V3[0](0.800000+0.800000=1.600000) / /V1[7]+V2[7]=V3[7](1.500000+0.100000=
1.600000) /
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/ejercicio8$ ./SumaVectores 11
Tiempo(seg.):0.000002202        / Tamaño Vectores:11   / V1[0]+V2[0]=V3[0](1.100000+1.100000=2.200000) / /V1[10]+V2[10]=V3[10](2.100000+0.1000
00=2.200000) /
```

. ¿Cuántos threads y cuántos cores como máximo podría utilizar la versión que ha implementado en el ejercicio 7? Razone su respuesta. ¿Cuántos threads y cuantos cores como máximo podría utilizar la versión que ha implementado en el ejercicio 8? Razone su respuesta.

RESPUESTA: En ninguna de las dos versiones hemos especificado el número de threads que se pueden usar, por lo que usará todos los disponibles, esto es, 4, el número de cores lógicos de la

máquina. En el segundo caso, como se definen las secciones, y cada sección ocupa un thread, el máximo que se utilizarán serán 4 (número máximo de secciones definidas a la vez).

. Rellenar una tabla como la Tabla 2 para atcgrid y otra para el PC local con los tiempos de ejecución de los programas paralelos implementados en los ejercicios 7 y 8 y el programa secuencial del Listado 1. Generar los ejecutables usando -O2. En la tabla debe aparecer el tiempo de ejecución del trozo de código que realiza la suma en paralelo (este es el tiempo que deben imprimir los programas). Ponga en la tabla el número de threads/cores que usan los códigos. Represente en una gráfica los tres tiempos. NOTA: Nunca ejecute en atcgrid código que imprima todos los componentes del resultado.

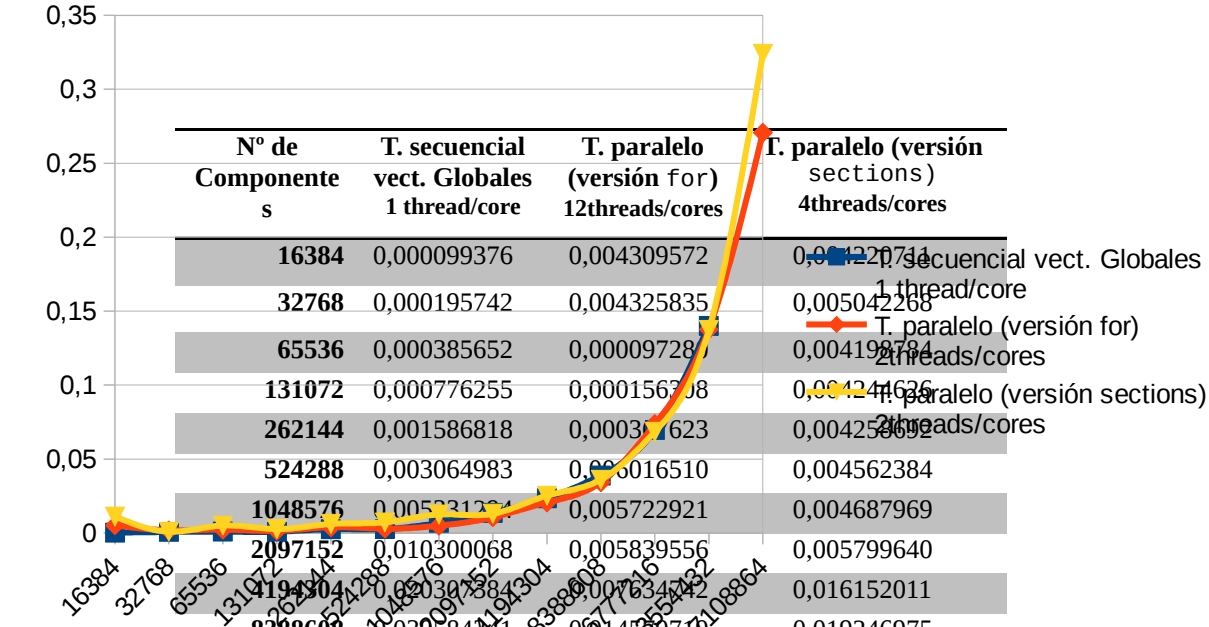
Tabla 2. Tiempos de ejecución de la versión secuencial de la suma de vectores y de las dos versiones paralelas. Sustituir en el encabezado de la tabla "¿?" por el número de threads utilizados, que debe coincidir con el número de cores físicos utilizados. PC LOCAL.

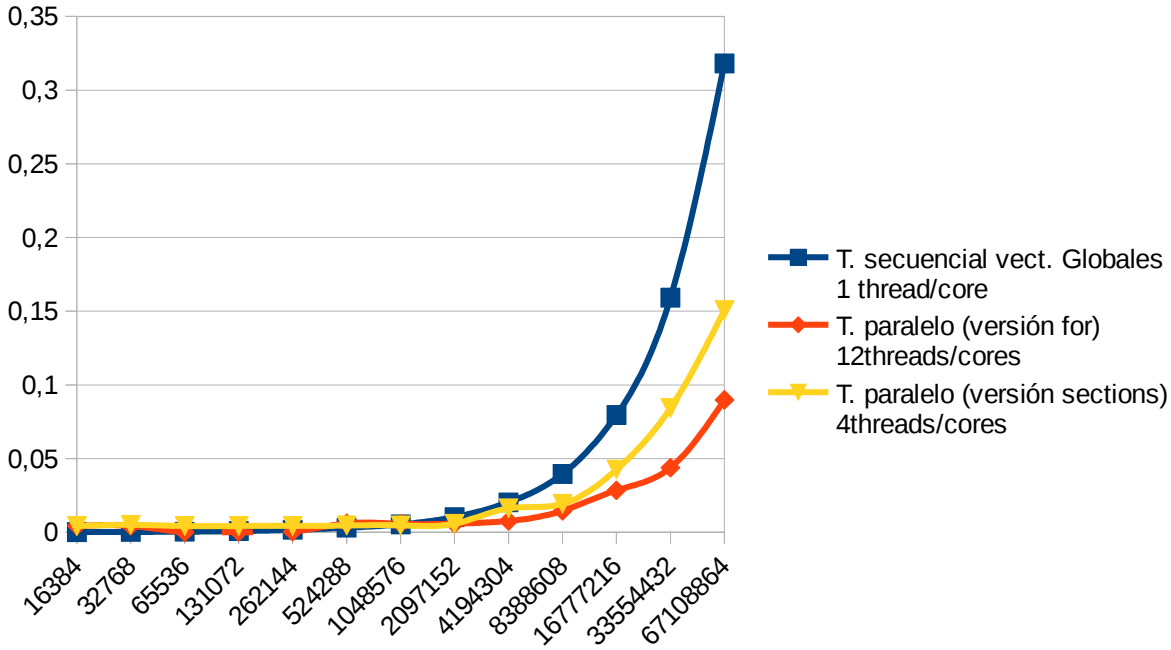
Nº de Componentes	**T. secuencial vect. Globales 1 thread/core**	**T. paralelo (versión `for`) 2threads/cores**	**T. paralelo (versión `sections`) 2threads/cores**
16384	0,000248464	0,005171004	0,011094365
32768	0,000713790	0,001769940	0,000908900
65536	0,000922976	0,002282706	0,005044120
131072	0,000879628	0,001463059	0,002661959
262144	0,002603906	0,003916882	0,006135333
524288	0,002672583	0,002923613	0,007188083
1048576	0,006781825	0,004888835	0,012486096
2097152	0,013622939	0,010620520	0,012990047
4194304	0,023546959	0,020619543	0,025205932
8388608	0,038866141	0,034253038	0,036003616
16777216	0,069613691	0,073644693	0,068444826
33554432	0,139954075	0,137662555	0,137327437
67108864	0.287331634	0,270596398	0,324083197

RESPUESTA:

Tabla 2.1 Tiempos de ejecución de la versión secuencial de la suma de vectores y de las dos versiones paralelas. Sustituir en el encabezado de la tabla "¿?" por el número de threads utilizados, que debe coincidir con el número de cores físicos utilizados. ATC GRID.

Nº de Componentes	T. secuencial vect. Globales 1 thread/core	T. paralelo (versión `for`) 12threads/cores	T. paralelo (versión `sections`) 4threads/cores
16384	0,000099376	0,004309572	0,004220711
32768	0,000195742	0,004325835	0,005042268
65536	0,000385652	0,000097287	0,004198784
131072	0,000776255	0,000156308	0,004241626
262144	0,001586818	0,000307623	0,004256692
524288	0,003064983	0,005016510	0,004562384
1048576	0,005831234	0,005722921	0,004687969
2097152	0,010300068	0,005839556	0,005799640
4194304	0,020307384	0,007634342	0,016152011
8388608	0,039584341	0,014397969	0,019246975
16777216	0,079561518	0,028438138	0,042537620
33554432	0,159174842	0,043806421	0,084261721
67108864	0,318160730	0,089795000	0,150674770





. Rellenar una tabla como la Tabla 3 para el PC local con el tiempo de ejecución, tiempo de CPU del usuario y tiempo CPU del sistema obtenidos con `time` para el ejecutable del ejercicio 7 y para el programa secuencial del Listado 1. Ponga en la tabla el número de threads/cores que usan los códigos. ¿El tiempo de CPU que se obtiene es mayor o igual que el tiempo real (*elapsed*)? Justifique la respuesta.
RESPUESTA: El tiempo que se obtiene es menor debido a que se utiliza la paralelización tanto en la creación de vectores como en el cálculo de la suma.

Tabla 3. Tiempos de ejecución de la versión secuencial de la suma de vectores y de las dos versiones paralelas. Sustituir en el encabezado de la tabla "¿?" por el número de threads utilizados.

Nº de Componentes	**Tiempo secuencial vect. Globales 1 thread/core**			**Tiempo paralelo/versión `for` 2 Threads/cores**		
	Elapsed	***CPU-user***	***CPU- sys***	***Elapsed***	***CPU-user***	***CPU- sys***
65536	0, 007s	0,105s	0,001s	0,005s	0,000s	0,008s
131072	0, 013s	0,188s	0,003s	0,007s	0,012s	0,000s
262144	0, 018s	0,235s	0,004s	0,007s	0,000s	0,016s
524288	0, 021s	0,203s	0,004s	0,010s	0,016s	0,012s
1048576	0, 021s	0,206s	0,028s	0,015s	0,036s	0,004s
2097152	0, 027s	0,245s	0,050s	0,025s	0,060s	0,016s
4194304	0, 031s	0,292s	0,115s	0,059s	0,116s	0,036s
8388608	0, 048s	0,335s	0,237s	0,084s	0,116s	0,104s
16777216	0, 082s	0,490s	0,507s	0,161s	0,380s	0,180s
33554432	0, 136s	0,725s	1,103s	0,340s	0,268s	0,268s
67108864	0, 139s	0,741s	1.137s	0,367s	0,592s	0,316s