

Universidad del Valle de Guatemala  
Facultad de Ingeniería  
Departamento de Ciencias de la Computación  
Programación paralela y distribuida  
Ing. Sebastián Galindo

## Laboratorio # 4



Christopher García 20541

Semestre II  
Octubre 2023

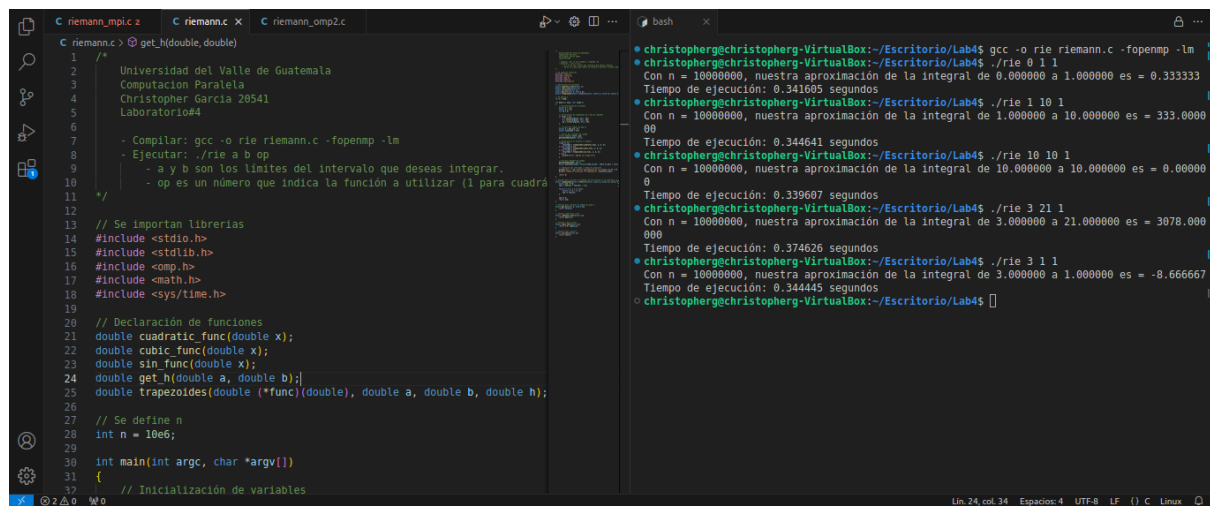
## Propuesta corto#3

- Fase 1: Inicializar valores ( $r=0$ )
  - El primer paso sería inicializar el entorno de Open MPI
  - El segundo paso sería definir en un método la función que se va a integrar recibiendo como parámetros  $a$ ,  $b$  (los límites)
    - `funcionAIntegrar(a,b):{}`
    - **Se agregaron más parámetros para que se pudieran hacer las sumas de Riemann**
- Fase 2: División de intervalos ( $r=0$ ) (Difusión)
  - En esta fase habrá un pasó y será el cálculo de subintervalos
  - Cada proceso calcula su subintervalo local  $[a\_local, b\_local]$  en función de su rango y el número total de procesos.
- Fase 3: Cálculo de sumas parciales ( $r=0-(n-1)$ )
  - En esta fase cada proceso realiza la suma de Riemann en su subintervalo local y obtiene una suma parcial local.
- Fase 4: Reducción de las sumas parciales ( $r=0$ )
  - En esta fase, se realizará una reducción de todas las sumas parciales locales para obtener la suma total de Riemann. Se utiliza una operación de reducción de MPI (i.e, `MPI_Reduce`) para sumar todas las sumas parciales locales y obtener la suma total.
- Fase 5: Cálculo del resultado final ( $r=0$ ) (Recolección)
  - Por último, en esta fase, el proceso 0 realiza el cálculo final de la integral sumando todas las sumas parciales reducidas y presentar un resultado final
- Fase 6: Finalización ( $r=0$ )
  - En esta fase, se liberan los recursos y se finaliza el entorno MPI.

## Datos generales para pruebas

- Se utilizó una función cuadrática
- Se utilizó  $n = 10^6$
- Se utilizó  $h$  como  $(b - a) / n$  (Esto proviene de la HT1)
- La máquina virtual cuenta con 4 procesadores

## Comparación#1 (Secuencial vs. OpenMP)



```
C riemann_mpic.z C riemann.c X C riemann_omp2.c bash X
C riemann.c > get_h(double,double)
1 /*
2  Universidad del Valle de Guatemala
3  Computación Paralela
4  Christopher Garcia 20541
5  Laboratorio#4
6
7  - Compilar: gcc -o rie riemann.c -fopenmp -lm
8  - Ejecutar: ./rie a b op
9    - a y b son los límites del intervalo que deseas integrar.
10   - op es un número que indica la función a utilizar (1 para cuadr
11 */
12
13 // Se importan librerías
14 #include <stdio.h>
15 #include <stdlib.h>
16 #include <omp.h>
17 #include <math.h>
18 #include <sys/time.h>
19
20 // Declaración de funciones
21 double quadratic_func(double x);
22 double cubic_func(double x);
23 double sin_func(double x);
24 double get_h(double a, double b);
25 double trapezoides(double (*func)(double), double a, double b, double h);
26
27 // Se define n
28 int n = 1000000;
29
30 int main(int argc, char *argv[])
31 {
32     // Inicialización de variables
```

```
christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$ gcc -o rie riemann.c -fopenmp -lm
christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$ ./rie 0 1 1
Con n = 10000000, nuestra aproximación de la integral de 0.000000 a 1.000000 es = 0.333333
Tiempo de ejecución: 0.341605 segundos
christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$ ./rie 1 10 1
Con n = 10000000, nuestra aproximación de la integral de 1.000000 a 10.000000 es = 333.0000
00
Tiempo de ejecución: 0.344641 segundos
christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$ ./rie 10 10 1
Con n = 10000000, nuestra aproximación de la integral de 10.000000 a 10.000000 es = 0.000000
00
Tiempo de ejecución: 0.339607 segundos
christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$ ./rie 3 21 1
Con n = 10000000, nuestra aproximación de la integral de 3.000000 a 21.000000 es = 3078.000
000
Tiempo de ejecución: 0.374626 segundos
christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$ ./rie 3 1 1
Con n = 10000000, nuestra aproximación de la integral de 3.000000 a 1.000000 es = -8.666667
Tiempo de ejecución: 0.344445 segundos
christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$
```

Imagen#1: Ejecución programa secuencial

```
riemann_mpi.c x riemann.c x riemann_omp2.c x
C riemann_omp2.c > | cat
1 /*
2  Universidad del Valle de Guatemala
3  Computacion Paralela
4  Christopher Garcia 20541
5  Laboratorio#4
6
7  - Compilar: gcc -o rie_omp riemann_omp2.c -fopenmp -lm
8  - Ejecutar: ./rie_omp a b num_threads op
9    - a y b son los limites del intervalo que deseas integrar.
10    - num_threads es el número de hilos con los que se trabajaran
11    - op es un número que indica la función a utilizar (1 para cuadrá
12 */
13
14 // Se importan librerias
15 #include <stdio.h>
16 #include <stdlib.h>
17 #include <omp.h>
18 #include <math.h>
19 #include <sys/time.h>
20
21 // Declaración de funciones
22 double quadratic_func(double x);
23 double cubic_func(double x);
24 double sin_func(double x);
25 double get_h(double a, double b);
26 double trapezoides(double (*func)(double), double a, double b, double h,
27
28 // Se define n
29 int n = 10e6;
30
31 int main(int argc, char *argv[])
32 {
33     if (argc < 4)
34     {
35         printf("Uso: ./rie_omp a b num_threads op\n");
36         return 1;
37     }
38     double a = atof(argv[1]);
39     double b = atof(argv[2]);
40     int num_threads = atoi(argv[3]);
41     char op = argv[4][0];
42     double result = 0.0;
43     double h = get_h(a, b);
44     double start_time = 0.0;
45     double end_time = 0.0;
46     double approx = 0.0;
47     double error = 0.0;
48     double exact = 0.0;
49     double error_percent = 0.0;
50     double error_absolute = 0.0;
51     double error_relative = 0.0;
52     double error_max = 0.0;
53     double error_min = 0.0;
54     double error_avg = 0.0;
55     double error_std = 0.0;
56     double error_var = 0.0;
57     double error_cov = 0.0;
58     double error_corr = 0.0;
59     double error_rho = 0.0;
60     double error_tau = 0.0;
61     double error_phi = 0.0;
62     double error_chi = 0.0;
63     double error_psi = 0.0;
64     double error_omega = 0.0;
65     double error_theta = 0.0;
66     double error_alpha = 0.0;
67     double error_beta = 0.0;
68     double error_gamma = 0.0;
69     double error_delta = 0.0;
70     double error_epsilon = 0.0;
71     double error_zeta = 0.0;
72     double error_eta = 0.0;
73     double error_theta = 0.0;
74     double error_iota = 0.0;
75     double error_kappa = 0.0;
76     double error_lambda = 0.0;
77     double error_mu = 0.0;
78     double error_nu = 0.0;
79     double error_xi = 0.0;
80     double error_omicron = 0.0;
81     double error_pi = 0.0;
82     double error_rho = 0.0;
83     double error_sigma = 0.0;
84     double error_tau = 0.0;
85     double error_upsilon = 0.0;
86     double error_phi = 0.0;
87     double error_chi = 0.0;
88     double error_psi = 0.0;
89     double error_omega = 0.0;
90     double error_theta = 0.0;
91     double error_alpha = 0.0;
92     double error_beta = 0.0;
93     double error_gamma = 0.0;
94     double error_delta = 0.0;
95     double error_epsilon = 0.0;
96     double error_zeta = 0.0;
97     double error_eta = 0.0;
98     double error_theta = 0.0;
99     double error_iota = 0.0;
100    double error_kappa = 0.0;
101    double error_lambda = 0.0;
102    double error_mu = 0.0;
103    double error_nu = 0.0;
104    double error_xi = 0.0;
105    double error_omicron = 0.0;
106    double error_pi = 0.0;
107    double error_rho = 0.0;
108    double error_sigma = 0.0;
109    double error_tau = 0.0;
110    double error_upsilon = 0.0;
111    double error_phi = 0.0;
112    double error_chi = 0.0;
113    double error_psi = 0.0;
114    double error_omega = 0.0;
115    double error_theta = 0.0;
116    double error_alpha = 0.0;
117    double error_beta = 0.0;
118    double error_gamma = 0.0;
119    double error_delta = 0.0;
120    double error_epsilon = 0.0;
121    double error_zeta = 0.0;
122    double error_eta = 0.0;
123    double error_theta = 0.0;
124    double error_iota = 0.0;
125    double error_kappa = 0.0;
126    double error_lambda = 0.0;
127    double error_mu = 0.0;
128    double error_nu = 0.0;
129    double error_xi = 0.0;
130    double error_omicron = 0.0;
131    double error_pi = 0.0;
132    double error_rho = 0.0;
133    double error_sigma = 0.0;
134    double error_tau = 0.0;
135    double error_upsilon = 0.0;
136    double error_phi = 0.0;
137    double error_chi = 0.0;
138    double error_psi = 0.0;
139    double error_omega = 0.0;
140    double error_theta = 0.0;
141    double error_alpha = 0.0;
142    double error_beta = 0.0;
143    double error_gamma = 0.0;
144    double error_delta = 0.0;
145    double error_epsilon = 0.0;
146    double error_zeta = 0.0;
147    double error_eta = 0.0;
148    double error_theta = 0.0;
149    double error_iota = 0.0;
150    double error_kappa = 0.0;
151    double error_lambda = 0.0;
152    double error_mu = 0.0;
153    double error_nu = 0.0;
154    double error_xi = 0.0;
155    double error_omicron = 0.0;
156    double error_pi = 0.0;
157    double error_rho = 0.0;
158    double error_sigma = 0.0;
159    double error_tau = 0.0;
160    double error_upsilon = 0.0;
161    double error_phi = 0.0;
162    double error_chi = 0.0;
163    double error_psi = 0.0;
164    double error_omega = 0.0;
165    double error_theta = 0.0;
166    double error_alpha = 0.0;
167    double error_beta = 0.0;
168    double error_gamma = 0.0;
169    double error_delta = 0.0;
170    double error_epsilon = 0.0;
171    double error_zeta = 0.0;
172    double error_eta = 0.0;
173    double error_theta = 0.0;
174    double error_iota = 0.0;
175    double error_kappa = 0.0;
176    double error_lambda = 0.0;
177    double error_mu = 0.0;
178    double error_nu = 0.0;
179    double error_xi = 0.0;
180    double error_omicron = 0.0;
181    double error_pi = 0.0;
182    double error_rho = 0.0;
183    double error_sigma = 0.0;
184    double error_tau = 0.0;
185    double error_upsilon = 0.0;
186    double error_phi = 0.0;
187    double error_chi = 0.0;
188    double error_psi = 0.0;
189    double error_omega = 0.0;
190    double error_theta = 0.0;
191    double error_alpha = 0.0;
192    double error_beta = 0.0;
193    double error_gamma = 0.0;
194    double error_delta = 0.0;
195    double error_epsilon = 0.0;
196    double error_zeta = 0.0;
197    double error_eta = 0.0;
198    double error_theta = 0.0;
199    double error_iota = 0.0;
200    double error_kappa = 0.0;
201    double error_lambda = 0.0;
202    double error_mu = 0.0;
203    double error_nu = 0.0;
204    double error_xi = 0.0;
205    double error_omicron = 0.0;
206    double error_pi = 0.0;
207    double error_rho = 0.0;
208    double error_sigma = 0.0;
209    double error_tau = 0.0;
210    double error_upsilon = 0.0;
211    double error_phi = 0.0;
212    double error_chi = 0.0;
213    double error_psi = 0.0;
214    double error_omega = 0.0;
215    double error_theta = 0.0;
216    double error_alpha = 0.0;
217    double error_beta = 0.0;
218    double error_gamma = 0.0;
219    double error_delta = 0.0;
220    double error_epsilon = 0.0;
221    double error_zeta = 0.0;
222    double error_eta = 0.0;
223    double error_theta = 0.0;
224    double error_iota = 0.0;
225    double error_kappa = 0.0;
226    double error_lambda = 0.0;
227    double error_mu = 0.0;
228    double error_nu = 0.0;
229    double error_xi = 0.0;
230    double error_omicron = 0.0;
231    double error_pi = 0.0;
232    double error_rho = 0.0;
233    double error_sigma = 0.0;
234    double error_tau = 0.0;
235    double error_upsilon = 0.0;
236    double error_phi = 0.0;
237    double error_chi = 0.0;
238    double error_psi = 0.0;
239    double error_omega = 0.0;
240    double error_theta = 0.0;
241    double error_alpha = 0.0;
242    double error_beta = 0.0;
243    double error_gamma = 0.0;
244    double error_delta = 0.0;
245    double error_epsilon = 0.0;
246    double error_zeta = 0.0;
247    double error_eta = 0.0;
248    double error_theta = 0.0;
249    double error_iota = 0.0;
250    double error_kappa = 0.0;
251    double error_lambda = 0.0;
252    double error_mu = 0.0;
253    double error_nu = 0.0;
254    double error_xi = 0.0;
255    double error_omicron = 0.0;
256    double error_pi = 0.0;
257    double error_rho = 0.0;
258    double error_sigma = 0.0;
259    double error_tau = 0.0;
260    double error_upsilon = 0.0;
261    double error_phi = 0.0;
262    double error_chi = 0.0;
263    double error_psi = 0.0;
264    double error_omega = 0.0;
265    double error_theta = 0.0;
266    double error_alpha = 0.0;
267    double error_beta = 0.0;
268    double error_gamma = 0.0;
269    double error_delta = 0.0;
270    double error_epsilon = 0.0;
271    double error_zeta = 0.0;
272    double error_eta = 0.0;
273    double error_theta = 0.0;
274    double error_iota = 0.0;
275    double error_kappa = 0.0;
276    double error_lambda = 0.0;
277    double error_mu = 0.0;
278    double error_nu = 0.0;
279    double error_xi = 0.0;
280    double error_omicron = 0.0;
281    double error_pi = 0.0;
282    double error_rho = 0.0;
283    double error_sigma = 0.0;
284    double error_tau = 0.0;
285    double error_upsilon = 0.0;
286    double error_phi = 0.0;
287    double error_chi = 0.0;
288    double error_psi = 0.0;
289    double error_omega = 0.0;
290    double error_theta = 0.0;
291    double error_alpha = 0.0;
292    double error_beta = 0.0;
293    double error_gamma = 0.0;
294    double error_delta = 0.0;
295    double error_epsilon = 0.0;
296    double error_zeta = 0.0;
297    double error
```

Imagen#2: Ejecución programa openMP

Límites de integral	Tiempo Secuencial	Tiempo OpenMP	Speedups
0, 1	0.341605 seg	0.340087 seg	1.000
1, 10	0.344641 seg	0.340123 seg	1.011
10, 10	0.339607 seg	0.355773 seg	0.957
3, 21	0.374626 seg	0.345042 seg	1.086
3, 1	0.344445 seg	0.336236 seg	1.024

## Comparación#2 (Secuencial vs. MPI)

```
riemann_mpic_z  riemann_x  riemann_omp_z
riemann.c> get_h(double, double)
1  /*
2   Universidad del Valle de Guatemala
3   Computacion Paralela
4   Christopher Garcia 20541
5   Laboratorio104
6
7   - Compilar: gcc -o rie riemann.c -fopenmp -lm
8   - Ejecutar: ./rie a b op
9   - a y b son los limites del intervalo que deseas integrar.
10  - op es un número que indica la función a utilizar (1 para cuadrática, 2 para
11  */
12
13  // Se importan librerías
14  #include <stdio.h>
15  #include <stdlib.h>
16  #include <omp.h>
17  #include <math.h>
18  #include <sys/time.h>
19
20  // Declaración de funciones
21  double quadratic_func(double x);
22  double cubic_func(double x);
23  double sin_func(double x);
24  double get_h(double a, double b);
25  double trapezoides(double (*func)(double), double a, double b, double h);
26
27  // Se define n
28  int n = 10e6;
29
30  int main(int argc, char *argv[])
31  {
32      // Inicialización de variables
33      double a, b, op;
34      if (argc < 4) {
35          printf("Uso: ./rie a b op\n");
36          return 1;
37      }
38      a = atof(argv[1]);
39      b = atof(argv[2]);
40      op = atoi(argv[3]);
41      if (op < 1 || op > 2) {
42          printf("Op debe ser 1 o 2\n");
43          return 1;
44      }
45      double h = get_h(a, b, op);
46      double suma = 0;
47      double t_inicio = clock();
48      for (int i = 0; i < n; i++) {
49          suma += func(a + i * h, op);
50      }
51      double t_fin = clock();
52      double tiempo = (t_fin - t_inicio) / CLOCKS_PER_SEC;
53      printf("Tiempo de ejecución: %f segundos\n", tiempo);
54      printf("Resultado: %f\n", suma * h);
55      return 0;
56  }
```

Imagen#3: Ejecución programa secuencial

```

C:riemann_mpi.c x C:riemann.c C:riemann_omp2.c ... bash x
C:riemann_mpi.c > ...
1 /*
2  Universidad del Valle de Guatemala
3  Computacion Paralela
4  Christopher Garcia 20541
5  Laboratorio#4
6
7  - Compilar: mpicc -o rie_mpi riemann_mpi.c -lm
8  - Ejecutar: mpirun -np 4 ./rie_mpi a b
9  | - a y b son los límites del intervalo que deseas integrar.
10 */
11
12 // Se importan librerías
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <mpi.h>
16 #include <math.h>
17 #include <sys/time.h>
18
19 // Declaración de funciones
20 double quadratic_func(double x);
21 double trapezoides(double (*func)(double), double a, double b, int n, double h);
22
23 // Número de subdivisiones (n)
24 int n = 10e6;
25
26 int main(int argc, char *argv[])
27 {
28     // Inicialización de MPI
29     MPI_Init(&argc, &argv);
30
31     int rank, size;
32     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
33 }

```

```

christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$ mpicc -o rie_mpi riema
nn mpi.c -lm
christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$ mpirun -np 4 ./rie_mpi
0 1
Con n = 10000000, nuestra aproximación de la integral de 0.000000 a 1.000000 e
s = 0.333333
Tiempo de ejecución: 0.034524 segundos
christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$ mpirun -np 4 ./rie_mpi
1 10
Con n = 10000000, nuestra aproximación de la integral de 1.000000 a 10.000000
es = 333.000000
Tiempo de ejecución: 0.041562 segundos
christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$ mpirun -np 4 ./rie_mpi
10 10
Con n = 10000000, nuestra aproximación de la integral de 10.000000 a 10.000000
es = 0.000000
Tiempo de ejecución: 0.041566 segundos
christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$ mpirun -np 4 ./rie_mpi
3 21
Con n = 10000000, nuestra aproximación de la integral de 3.000000 a 21.000000
es = 3078.000000
Tiempo de ejecución: 0.035660 segundos
christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$ mpirun -np 4 ./rie_mpi
3 1
Con n = 10000000, nuestra aproximación de la integral de 3.000000 a 1.000000 e
s = -8.666667
Tiempo de ejecución: 0.048318 segundos
christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$

```

Imagen#4: Ejecución programa MPI

Límites de integral	Tiempo Secuencial	Tiempo MPI	Speedups
0, 1	0.346387 seg	0.034524 seg	10.033
1, 10	0.353080 seg	0.041562 seg	8.495
10, 10	0.342327 seg	0.041456 seg	8.258
3, 21	0.345477 seg	0.035660 seg	9.688
3, 1	0.347459 seg	0.048318 seg	7.191

### Comparación#3 (OpenMP vs. MPI)

```

C:riemann_mpi.c x C:riemann_omp2.c x ... bash x
C:riemann_omp2.c > ...
1 /*
2  Universidad del Valle de Guatemala
3  Computacion Paralela
4  Christopher Garcia 20541
5  Laboratorio#4
6
7  - Compilar: gcc -o rie_omp riemann_omp2.c -fopenmp -lm
8  - Ejecutar: ./rie_omp #b num_threads op
9  | - a y b son los límites del intervalo que deseas integra
10 | - num_threads es el número de hilos con los que se traba
11 | - op es un número que indica la función a utilizar (1 pa
12 */
13
14 // Se importan librerías
15 #include <stdio.h>
16 #include <stdlib.h>
17 #include <omp.h>
18 #include <math.h>
19 #include <sys/time.h>
20
21 // Declaración de funciones
22 double quadratic_func(double x);
23 double cubic_func(double x);
24 double sin_func(double x);
25 double get_h(double a, double b);
26 double trapezoides(double (*func)(double), double a, double b, d
27
28 // Se define n
29 int n = 10e6;
30
31 int main(int argc, char *argv[])
32 {
33 }

```

```

christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$ gcc -o rie_omp riemann_omp2.c -fopenmp -lm
christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$ ./rie_omp 0 1 1 1
Con n = 10000000, nuestra aproximación de la integral de 0.000000 a 1.000000 es = 0.333333
Tiempo de ejecución: 0.344863 segundos
christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$ ./rie_omp 1 10 1 1
Con n = 10000000, nuestra aproximación de la integral de 1.000000 a 10.000000 es = 333.000000
Tiempo de ejecución: 0.343575 segundos
christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$ ./rie_omp 10 10 1 1
Con n = 10000000, nuestra aproximación de la integral de 10.000000 a 10.000000 es = 0.000000
Tiempo de ejecución: 0.343707 segundos
christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$ ./rie_omp 3 21 1 1
Con n = 10000000, nuestra aproximación de la integral de 3.000000 a 21.000000 es = 3078.000000
Tiempo de ejecución: 0.367982 segundos
christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$ ./rie_omp 3 1 1 1
Con n = 10000000, nuestra aproximación de la integral de 3.000000 a 1.000000 es = -8.666667
Tiempo de ejecución: 0.348198 segundos
christopherg@christopherg-VirtualBox:~/Escritorio/Lab4$

```

Imagen#5: Ejecución programa openMP

```

1  /*
2  Universidad del Valle de Guatemala
3  Computación Paralela
4  Christopher Garcia 20541
5  Laboratorio#4
6
7  - Compilar: mpicc -o rie_mpi riemann_mpi.c -lm
8  - Ejecutar: mpirun -np 4 ./rie_mpi a b
9  - a y b son los límites del intervalo que deseas integrar.
10 */
11
12 // Se importan librerías
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <mpi.h>
16 #include <math.h>
17 #include <sys/time.h>
18
19 // Declaración de funciones
20 double quadratic_func(double x);
21 double trapezoides(double (*func)(double), double a, double b, int
22
23 // Número de subdivisiones (n)
24 int n = 10e6;
25
26 int main(int argc, char *argv[])
27 {
28     // Inicialización de MPI
29     MPI_Init(&argc, &argv);
30
31     int rank, size;
32     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
33 }

```

```

christopher@christopher-VirtualBox:~/Escritorio/Lab4$ mpicc -o rie_mpi riemann_mpi.c -lm
christopher@christopher-VirtualBox:~/Escritorio/Lab4$ mpirun -np 4 ./rie_mpi 0 1
Con n = 10000000, nuestra aproximación de la integral de 0.000000 a 1.000000 es = 0.333333
Tiempo de ejecución: 0.029727 segundos
christopher@christopher-VirtualBox:~/Escritorio/Lab4$ mpirun -np 4 ./rie_mpi 1 10
Con n = 10000000, nuestra aproximación de la integral de 1.000000 a 10.000000 es = 333.000000
Tiempo de ejecución: 0.030007 segundos
christopher@christopher-VirtualBox:~/Escritorio/Lab4$ mpirun -np 4 ./rie_mpi 10 10
Con n = 10000000, nuestra aproximación de la integral de 10.000000 a 10.000000 es = 0.000000
Tiempo de ejecución: 0.035542 segundos
christopher@christopher-VirtualBox:~/Escritorio/Lab4$ mpirun -np 4 ./rie_mpi 3 21
Con n = 10000000, nuestra aproximación de la integral de 3.000000 a 21.000000 es = 3078.000000
Tiempo de ejecución: 0.030675 segundos
christopher@christopher-VirtualBox:~/Escritorio/Lab4$ mpirun -np 4 ./rie_mpi 3 1
Con n = 10000000, nuestra aproximación de la integral de 3.000000 a 1.000000 es = -8.666667
Tiempo de ejecución: 0.031964 segundos
christopher@christopher-VirtualBox:~/Escritorio/Lab4$

```

Imagen#6: Ejecución programa MPI

Límites de integral	Tiempo OpenMP	Tiempo MPI	Speedups
0, 1	0.344863 seg	0.029727 seg	11.601
1, 10	0.343575 seg	0.030007 seg	11.446
10, 10	0.345707 seg	0.035542 seg	9.727
3, 21	0.367982 seg	0.030675 seg	11.996
3, 1	0.348198 seg	0.031964 seg	10.893

## Discusión y conclusión de resultados

### - Comparación#1 (Secuencial vs. OpenMP)

En esta primera comparación, observamos que los speedups oscilan alrededor de 1. Esto sugiere que la versión paralela con OpenMP no proporciona una mejora significativa en el rendimiento en comparación con la versión secuencial. El valor promedio de speedup es aproximadamente 1.016, lo que indica que, en promedio, OpenMP no acelera significativamente la ejecución del programa en este conjunto de datos o carga de trabajo específica. Estos resultados podrían deberse a varias razones, como una implementación ineficiente de OpenMP en el programa, una carga de trabajo que no se beneficia de la paralelización, o la naturaleza del algoritmo en sí (que no utiliza bien los hilos indicados).

### - Comparación#2 (Secuencial vs. MPI)

En la segunda comparación, los speedups son más notables en comparación con la versión secuencial, con valores que oscilan entre 7.191 y 10.033. Esto sugiere que la implementación con Open MPI ofrece una mejora significativa en el rendimiento en comparación con la versión secuencial. El valor promedio de speedup es aproximadamente 8.735. Estos resultados son prometedores y muestran que la paralelización con Open MPI ha tenido un impacto positivo en la eficiencia del programa. Sin embargo, siempre es útil analizar si hay margen para una mejora adicional. También es importante tener en cuenta

que la eficiencia de MPI a menudo depende de la arquitectura del clúster y la comunicación entre nodos.

### - Comparación#3 (OpenMP vs. MPI)

En la última comparación, se compara el rendimiento entre las implementaciones paralelas utilizando OpenMP y MPI. Los speedups son notables, con valores promedio de aproximadamente 10.932. Esto indica que ambas implementaciones son eficientes en términos de velocidad, pero MPI tiende a ser ligeramente más rápido en este conjunto de datos específico. Los resultados de esta comparación indican que tanto OpenMP como MPI son enfoques válidos para la paralelización. Sin embargo, la elección entre ellos podría depender de factores como la arquitectura del sistema, la naturaleza del algoritmo y la complejidad de la programación paralela.

En resumen, los resultados sugieren que la elección entre OpenMP y Open MPI debe basarse en la naturaleza específica del problema y la plataforma de hardware en la que se está ejecutando. Mientras que OpenMP parece no ofrecer una mejora significativa en este caso particular, Open MPI ha demostrado ser eficiente en la aceleración del programa. Sin embargo, es fundamental seguir evaluando y optimizando las implementaciones para lograr un mejor rendimiento. Además, en futuros trabajos, se podría considerar explorar otras estrategias de paralelización y medir su impacto en el rendimiento.

Link del repositorio: [https://github.com/ChristopherG19/UVG\\_Paralela\\_Lab4.git](https://github.com/ChristopherG19/UVG_Paralela_Lab4.git) \*\*

**\*\* (En cada uno de los archivos, en el encabezado, se adjuntan los comandos para compilar y ejecutar los mismos)**