



# Computación Paralela y Distribuida

2022-II

---

José Fiestas

26/09/22

Universidad de Ingeniería. Tecnología  
jfiestas@utec.edu.pe

# Unidad 4: Comunicación y coordinación

## Objetivos:

1. Pasos de Mensaje: MPI, Mensajes Punto a Punto, MPI, Comunicación Colectiva, Blocking vs non-blocking
2. Memoria Compartida: OMP, Constructores y cláusulas
3. Programacion Híbrida

## OpenMP (Open Multi-processing)

# Uso de memoria en OPENMP:

Despues de Print 1, el valor de x puede ser 2 o 5, dependiendo del tiempo de ejecución de los hilos (threads) y de como se escribe en x, que puede NO ser 5 en Print1, ya que este puede ejecutarse antes de asignar el valor de x

---

```
1  #include <stdio.h>
2  #include <omp.h>
3  int main(){
4  int x; x = 2;
5  #pragma omp parallel num_threads(2) shared(x) {
6  if (omp_get_thread_num() == 0) { x = 5;
7  } else {
8  /* Print 1: lectura concurrente de x */
9  printf("1: hilo %d: x = %d\n", omp_get_thread_num(), x ); }
10 if (omp_get_thread_num() == 0) {
11 /* Print 2 */
12 printf("2: hilo %d: x = %d\n", omp_get_thread_num(), x );
13 } else {
14 /* Print 3 */
15 printf("3: hilo %d: x = %d\n", omp_get_thread_num(), x );
16 } }
17 return 0; }
```

---

# Uso de memoria en OPENMP:

Por ello, la barrera `#pragma omp barrier` luego de `Print1` contiene flushes implícitos en todos los threads, para garantizar que el valor 5 sea impreso por `Print2` y `Print3`

---

```
1  #include <stdio.h>
2  #include <omp.h>
3  int main(){
4  int x; x = 2;
5  #pragma omp parallel num_threads(2) shared(x) {
6  if (omp_get_thread_num() == 0) { x = 5;
7  } else {
8  /* Print 1: lectura concurrente de x */
9  printf("1: hilo %d: x = %d\n", omp_get_thread_num(), x ); }
10 if (omp_get_thread_num() == 0) {
11 /* Print 2 */
12 printf("2: hilo %d: x = %d\n", omp_get_thread_num(), x );
13 } else {
14 /* Print 3 */
15 printf("3: hilo %d: x = %d\n", omp_get_thread_num(), x );
16 } }
17 return 0; }
```

---

# Variables de control interno

Cada tarea generada en una región en paralelo recibe una copia de ICVs, que controlan el comportamiento del programa.

OMP\_SET\_NUM\_THREADS: define el número de threads que serán usados en la region que sigue

---

```
1 #include <omp.h>
2 void omp_set_num_threads(int num_threads)
```

---

OMP\_SET\_MAX\_ACTIVE\_LEVELS: Limita el número de regiones en paralelo anidadas

---

```
1 #include <omp.h>
2 void omp_set_max_active_levels (int max_levels)
```

---

Cada tarea generada en una región en paralelo recibe una copia de ICVs, que controlan el comportamiento del programa.

OMP\_SET\_NESTED: Activa/desactiva paralelismo anidado

---

```
1 #include <omp.h>
2 void omp_set_nested(int nested)
```

---

OMP\_SET\_DYNAMIC: Activa/desactiva ajuste dinámico del número de threads accesibles en la región en paralelo

---

```
1 #include <omp.h>
2 void omp_set_dynamic(int dynamic_threads)
```

---

La region en paralelo externa crea grupos de 2 threads que van a ejecutar la tarea. Cada tarea un grupo de 3 threads, cada uno ejecutando una tarea y asignando 4 a nthreads

---

```
1  int main (void) {
2      omp_set_nested(1);
3      omp_set_max_active_levels(8);
4      omp_set_dynamic(0);
5      omp_set_num_threads(2);
6      #pragma omp parallel {
7          omp_set_num_threads(3);
8          #pragma omp parallel {
9              omp_set_num_threads(4);
10             #pragma omp single {
11                 printf ("Inner: max_act_lev=%d, num_thds=%d, max_thds=%d\n",
12                     omp_get_max_active_levels(), omp_get_num_threads(),
13                     omp_get_max_threads());
14             }
15         }
```

---



El print de la region externa se ejecuta solo por uno de los threads del grupo, mientras que el print de la región interna hace lo mismo 2 veces, ya que existen dos threads en esta región en paralelo

---

```
1 #pragma omp barrier
2 #pragma omp single
3 {
4     printf ("Outer: max_act_lev=%d, num_thds=%d, max_thds=%d\n",
5         omp_get_max_active_levels(), omp_get_num_threads(), omp_get_max_threads())
6 }
7 }
8 return 0;
9 }
```

---

# Constructor parallel

Se puede usar en algoritmos paralelos de granularidad gruesa (coarse grain parallel programmes) En el ejemplo, cada thread en la región en paralelo, decide en que parte del array x va a trabajar, basada en el número total de threads

---

```
1  #include <omp.h>
2  void subdomain(float *x, int istart, int ipoints)
3  {
4      int i;
5      for (i = 0; i < ipoints; i++)
6          x[istart+i] = 123.456;
7  }
8  void sub(float *x, int npoints)
9  {
10     int iam, nt, ipoints, istart;
11     #pragma omp parallel default(shared)
12     private(iam,nt,ipoints,istart)
13     {
14         iam = omp_get_thread_num();
15         nt = omp_get_num_threads();
```

---

# Constructor parallel

Se puede usar en algoritmos paralelos de granularidad gruesa (coarse grain parallel programmes) En el ejemplo, cada thread en la región en paralelo, decide en que parte del array  $x$  va a trabajar, basada en el número total de threads

---

```
1  if (iam == nt-1) /* last thread may do more */
2  ipoints = npoints - istart;
3  subdomain(x, istart, ipoints);  }
4  }
5  int main()
6  {
7  float array[10000];
8  sub(array, 10000);
9  return 0;
10 }
```

---

# Constructor sections

La cláusula `firstprivate` inicializa la copia privada de `section_count` en cada hilo. Las secciones modifican `section_count`. Si son ejecutadas por hilos distintos, cada uno imprimirá el valor 1. Si el mismo hilo ejecuta ambas secciones, una imprimirá 1 y la otra, 2.

```
1  int main( ) {
2      int section_count = 0;
3      omp_set_dynamic(0);
4      omp_set_num_threads(NT);
5      #pragma omp parallel
6      #pragma omp sections firstprivate( section_count )
7      {
8          #pragma omp section
9          {
10             section_count++;
11             printf( "section_count %d\n", section_count );
12         }
13         #pragma omp section
14         {
15             section_count++;
16             printf( "section_count %d\n", section_count );
17         }
18     }
19 }
```

# Cláusula nowait

En el ejemplo, static schedule distribuye la misma secuencia numérica entre los threads que ejecutan los 3 ciclos (loops) en paralelo. Esto permite el uso de nowait, incluso cuando hay una dependencia entre los ciclos. Cada hilo ejecuta la misma secuencia numérica en cada ciclo. El número de iteraciones es el mismo en cada ciclo.

---

```
1 void nowait_example2
2 (int n, float *a, float *b, float *c, float *y, float *z)
3 {
4     int i;
5     #pragma omp parallel
6     {
7         #pragma omp for schedule(static) nowait
8         for (i=0; i<n; i++)
9             c[i] = (a[i] + b[i]) / 2.0f;
10        #pragma omp for schedule(static) nowait
11        for (i=0; i<n; i++)
12            z[i] = sqrtf(c[i]);
13        #pragma omp for schedule(static) nowait
14        for (i=1; i<=n; i++)
15            y[i] = z[i-1] + a[i];
16    }
17 }
```

# Cláusula collapse

Las iteraciones k y j se colapsan en un bucle con un espacio de iteración mas grande, que es dividido entre los threads La secuencia de ejecución de las iteraciones k y j determinan el orden de iteración en el bucle colapsado. Esto implica que el último valor de k será 2 y de j será 3. klast y jlast son lastprivate. Este ejemplo imprime: 2 3

---

```
1  #include <stdio.h>
2  void test()
3  {
4  int j, k, jlast, klast;
5  #pragma omp parallel
6  {
7  #pragma omp for collapse(2) lastprivate(jlast, klast)
8  for (k=1; k<=2; k++)
9  for (j=1; j<=3; j++)
10 {
11 jlast=j;
12 klast=k;
13 }
14 #pragma omp single
15 printf("%d %d\n", klast, jlast);
16 }
17 }
```

# Constructor single

En el ejemplo, solo un thread imprime cada uno de los mensajes de avance del trabajo. El resto de threads evitará la region single y se detendrá en la barrera implícita al final del constructor single. En el último trabajo se puede continuar sin necesidad de esperar (cláusula `nowait`).

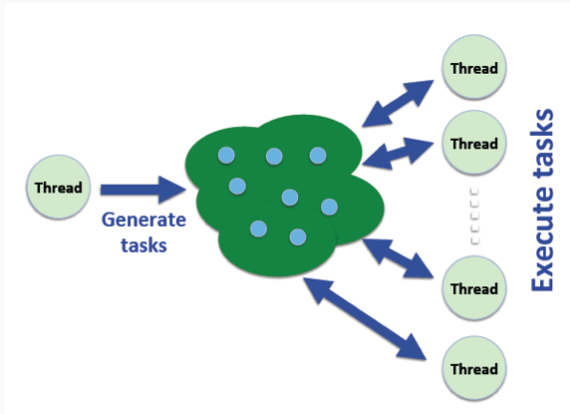
---

```
1  #include <stdio.h>
2  void work1() {}
3  void work2() {}
4  void single_example()
5  {
6  #pragma omp parallel
7  {
8  #pragma omp single
9  printf("Beginning work1.\n");
10 work1();
11 #pragma omp single
12 printf("Finishing work1.\n");
13 #pragma omp single nowait
14 printf("Finished work1 and beginning work2.\n");
15 work2();
16 }
17 }
```

---

# Constructor task

Task define una tarea, que puede ser ejecutada por el thread que la encuentre, o diferida a otro thread. Su ejecución puede ser controlada por directivas.





# Constructor task

Task define una tarea, que puede ser ejecutada por el thread que la encuentre, o diferida a otro thread. Su ejecución puede ser controlada por directivas.

---

```
1  int main(){
2
3  #pragma omp parallel num_threads(2)
4  {
5  #pragma omp single
6  {
7  #pragma omp task
8  printf("t1: impresion random desde el hilo: %d\n", omp_get_thread_num());
9  #pragma omp task
10 printf("t2: impresion random desde el hilo: %d\n", omp_get_thread_num());
11 #pragma omp task
12 printf("t3: impresion random desde el hilo: %d\n", omp_get_thread_num());
13 }
14 }
15 }
```

---

# Constructor task: cláusula depend

Dependencia tipo **in**: la tarea depende de todas las tareas antes generadas con al menos uno de sus elementos de lista en una cláusula out o inout

Dependencia tipo **out** y **inout**: la tarea depende de todas las tareas antes generadas con al menos uno de sus elementos de lista en una cláusula in, out o inout

---

```
1 #pragma omp task depend(dependency-type: list)
2   ... structured block ...
```

---

# Constructor task: cláusula depend

La cláusula depend condiciona el orden de tareas (tasks). El tipo puede ser in, out, inout- La dependencia se da por las referencias a variables dadas por tasks anteriores. El código imprime "x=2", ya que la dependencia se da antes de la impresión

---

```
1  int main()
2  {
3      int x = 1;
4      #pragma omp parallel
5      #pragma omp single
6      {
7          #pragma omp task shared(x) depend(out: x)
8              x = 2;
9          #pragma omp task shared(x) depend(in: x)
10             printf("x = %d\n", x);
11     }
12     return 0;
13 }
```

---

# Constructor task: cláusula depend

El código imprimira "x=1". Ya que la dependencia con x=2 esta despues de la impresión

---

```
1  int main()
2  {
3      int x = 1;
4      #pragma omp parallel
5      #pragma omp single
6      {
7          #pragma omp task shared(x) depend(in: x)
8          printf("x = %d\n", x);
9          #pragma omp task shared(x) depend(out: x)
10         x = 2;
11     }
12     return 0;
13 }
```

---

# Constructor task: cláusula depend

El código imprimirá "x=2", ya que la dependencia con x=2 se da al final de la secuencia de tasks, al tener que esperar que ambos se ejecuten (debido al taskwait)

---

```
1  int main()
2  {
3      int x;
4      #pragma omp parallel
5      #pragma omp single
6      {
7          #pragma omp task shared(x) depend(out: x)
8              x = 1;
9          #pragma omp task shared(x) depend(out: x)
10             x = 2;
11         #pragma omp taskwait
12         printf("x = %d\n", x);
13     }
14     return 0;
15 }
```

---

# Constructor task: competencia de dependencias

Potencial concurrencia de ejecución de tareas. Las dos últimas tareas son dependientes de la primera. Pero no hay dependencia entre estas, que pueden ejecutarse en cualquier orden, o en competencia, si está disponible mas de un thread. Posibles salidas son: " $x+1 = 3$ ,  $x+2 = 4$ ", " $x+2 = 4$ ,  $x+1 = 3$ "

---

```
1  #include <stdio.h>
2  int main() {
3      int x = 1;
4      #pragma omp parallel
5      #pragma omp single
6      {
7          #pragma omp task shared(x) depend(out: x)
8          x = 2;
9          #pragma omp task shared(x) depend(in: x)
10         printf("x + 1 = %d. ", x+1);
11         #pragma omp task shared(x) depend(in: x)
12         printf("x + 2 = %d\n", x+2);
13     }
14 }
```

---

# Constructor taskgroup

Permite agrupar y sincronizar tareas (tasks). En el ejemplo, un task se crea en la región paralela, y luego un recorrido de árbol en paralelo es creado (compute\_tree()). Taskgroup asegura que start\_background\_work() no participe de la sincronización y se pueda ejecutar en paralelo libremente. Lo contrario pasa con taskwait (donde todas las tareas participan de la sincronización)

---

```
1  int main() {
2  int i;
3  init_tree(tree);
4  #pragma omp parallel
5  #pragma omp single {
6  #pragma omp task
7  start_background_work();
8  for (i = 0; i < max_steps; i++) {
9  #pragma omp taskgroup {
10 #pragma omp task
11 compute_tree(tree);
12 } // wait on tree traversal in this step
13 }} // background work required
14 print_results();
15 }
```

# Constructor taskyield






Los tasks ejecutan `something_useful()`, para luego hacer cálculos en una región crítica. Taskyield permite suspender el task actual si es que no puede acceder a la región crítica, y ejecutar otro task en vez de ese.

---

```
1  #include <omp.h>
2  void something_useful ( void );
3  void something_critical ( void );
4  void foo ( omp_lock_t * lock, int n )
5  {
6      int i;
7      for ( i = 0; i < n; i++ )
8          #pragma omp task
9          {
10             something_useful();
11             while ( !omp_test_lock(lock) ) {
12                 #pragma omp taskyield
13             }
14             something_critical();
15             omp_unset_lock(lock);
16         }
17     }
```

---



-  David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 2nd. Morgan Kaufmann, 2013. isbn: 978-0-12-415992-1.
-  Norm Matloff. *Programming on Parallel Machines*. University of California, Davis, 2014.
-  Peter S. Pacheco. *An Introduction to Parallel Programming*. 1st. Morgan Kaufmann, 2011. isbn: 978-0-12-374260- 5.
-  Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. 1st. McGraw-Hill Education Group, 2003. isbn: 0071232656.
-  Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Program- ming*. 1st. Addison-Wesley Professional, 2010. isbn: 0131387685, 9780131387683.