Serial execution

The execution time of a program with N instructions on a processor that is able to execute F instructions per second is

$$T = N \div F$$

One could execute the program faster (i.e. reduce T) by augmenting the value of F. And this has been the trend during more than 30 years of technology and computer architecture evolution.

Parallel execution

Ideally, each processor could receive $\frac{1}{P}$ of the program, reducing its execution time by P

$$T = (N \div P) \div F$$

Need to manage and coordinate the execution of tasks, ensuring correct access to shared resources

Throughput

Multiple programs executing at the same time on multiple processors, k programs on P processors, each program receives P/k processors.

Condición de carrera - data race

- Se da sobre las variables compartidas.
- Todos los threads acceden y modifican "a la vez", cualquier resultado es posible.
- La solución pasa por **sincronizar las operaciones de escritura (con locks)** o usar **operaciones atómicas**.

Inanición - starvation

- Se produce cuando un thread **no puede acceder a un recurso compartido**, siempre esta ocupado.
- El thread se bloquea durante mucho tiempo y no puede avanzar.
- No hay solución, se puede mitigar **balanceando la carga de trabajo** entre los threads.

Abrazo mortal - dead lock

- Dos o más threads se esperan mutuamente.
- El programa se queda en una espera infinita.
- El programado debe controlar el orden de las reservas de los recursos.

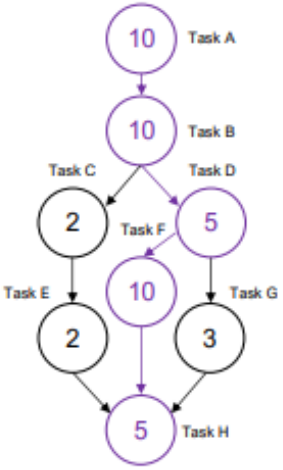
Live lock

- Variante del dead-lock; los threads no se esperan mutuamente sino que saltan de estado continuamente

Task dependence graph

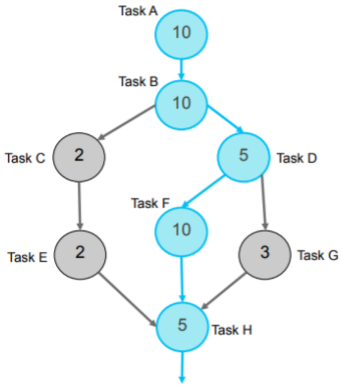
Graphical representation of the decomposition.

- Directed Acyclic Graph.
- Node = task, its weight represents the amount of work be done.
- Edge = dependence, each successor node can only execute after predecessor node completed.



Parallel machine abstraction:

- P identical processors
- Each processor executes a node at a time
- $T_1 = \sum_{i=1}^{nodes}(work\_node_i)$ = sum of all node's weights
- **Critical path**: path in the task graph with the highest accumulated work.
- $T_\infty = \sum_{i \in critical\_path}(wor\_node_i)$ = sum of all node's weights of critical path assuming sufficient processors.
- $Parallelism = T_1/T_\infty$ if sufficient processors were available.
- P_{min} is the minimum number of processors necessary to achieve *Parallelism*.



- $T_1 = 47$ including tasks {ABCDEFGH}
- Possible paths:

  {ABCEH} with total cost 29
  {ABDFH} with total cost 40
  {ABDGH} with total cost 33

- $T_\infty = 40$ for critical path {ABDFH}
- $Parallelism = 47/40 = 1.175$
- $P_{min} = 2$

Granularity and parallelism

The granularity of the task decomposition is determined by the computational size of the nodes (tasks) in the task graph.

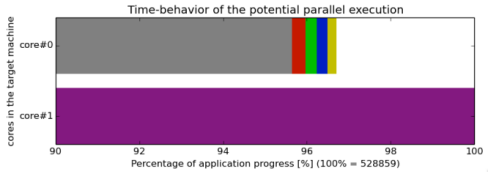
	Coarse grain	Fine grain	Medium grain
Number of tasks	1	n	n/m
Iterations per task	n	1	m
Parallelism	1	n	n/m

```
count = 0;
for ( i=0 ; i< n ; i++ )
  if (X[i].Color == "Green") count++;
```

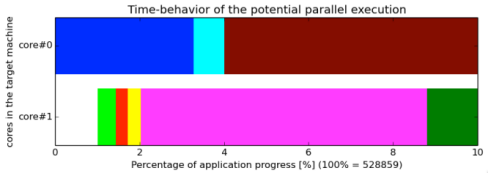
Coarse-grain decomposition: The whole loop is a task

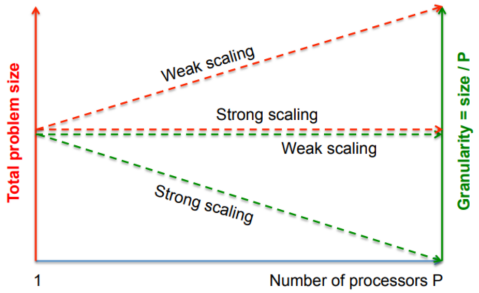
Fine-grain decomposition: Each iteration of the loop is a task

- Load unbalance



- Task creation overhead





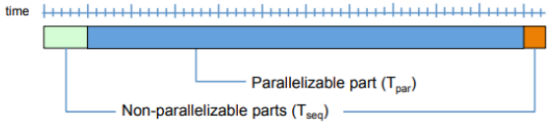
Strong vs. weak scalability

Two usual scenarios to evaluate the scalability of one application:

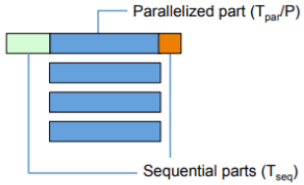
- Increase the number of processors P with constant problem size (strong scaling $\rightarrow$ reduce the execution time)
- Increase the number of processors P with problem size proportional to P (weak scaling $\rightarrow$ solve larger problem)

Amdahl's law

Assume the following simplified case, where the parallel fraction φ is the fraction, of total execution time, the program can be parallelized



$$T_1 = T_{seq} + T_{par}$$
$$\varphi = T_{par}/T_1$$
$$T_{seq} = (1-\varphi) \times T_1$$
$$T_{par} = \varphi \times T_1$$



$$T_1 = (1-\varphi) \times T_1 + \varphi \times T_1$$

$$T_P = T_{seq} + T_{par}/P$$
$$T_P = (1-\varphi) \times T_1 + (\varphi \times T_1/P)$$

From where we can compute the speed–up S_P that can be achieved as

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{(1-\varphi) \times T_1 + (\varphi \times T_1/P)}$$

$$S_p = \frac{1}{((1-\varphi) + \varphi/P)}$$

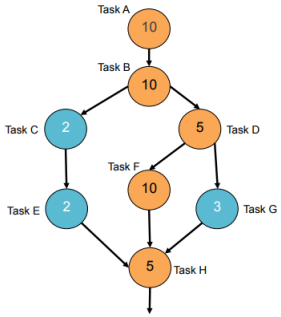
Two particular cases:

$$\varphi = 0 \rightarrow S_p = 1$$
$$\varphi = 1 \rightarrow S_p = P$$

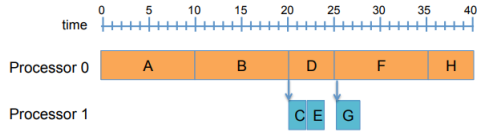
When $P \rightarrow \infty$ the expression of the speed–up becomes

$$S_{p\rightarrow\infty} = \frac{1}{(1-\varphi)}$$

Execution time bounds on P processors



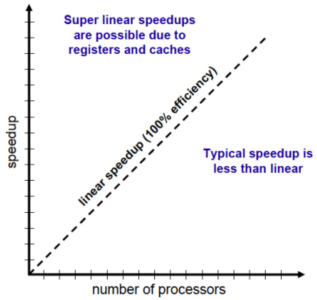
- T_p = execution time on P processors
- **Task scheduling**: how are tasks assigned to processors? For example:



- Lower bounds
  - $T_p \geq T_1/P$
  - $T_p \geq T_\infty$

Speedup S_p: relative reduction of the sequential execution time when using P processors

- $S_p = T_1 \div T_p$
- In this example: $T_2 = 40,\ S_2 = 47/40 = 1.175$



- Scalability: how the speed-up evolves when the number of processors is increased
- Efficiency: $E_p = S_p \div P$

Amdahl's law (with constant and linear overheads)

$$T_p = (1 - \varphi) \times T_1 + \varphi \times T_1/p + overhead(p)$$

Other sources of overhead

- **Data sharing:** can be explicit via messages, or implicit via a memory hierarchy (caches)
- **Idleness:** thread cannot find any useful work to execute (e.g. dependences, load imbalance, poor communication and computation overlap or hiding of memory latencies, ...)
- **Computation:** extra work added to obtain a parallel algorithm (e.g. replication)
- **Memory:** extra memory used to obtain a parallel algorithm (e.g. impact on memory hierarchy, ...)
- **Contention:** competition for the access to shared resources (e.g. memory, network)

How to model data sharing overhead?

- Processors access to local data (in its own memory) using regular load/store instructions
- We will assume that local accesses take zero overhead.
- Processors can access remote data (in other processors) using a message-passing model (remote load instruction[2])
- To model the time needed to access remote data we will use two components:
  - Start up: time spent in preparing the remote access (t_s)
  - Transfer: time spent in transferring the message (number of bytes m, time per byte t_w) from/to the remote location

$$t_{access} = t_s + m \times t_w$$

- Synchronization between the two processors involved may be necessary to guarantee that the data is available

Estratègies de descomposició de dades d'una matriu a calcular de mida NxN

Per files o columnes:
- Cada processador processa segments de N*N/P elements de la matriu.
- Útil quan hi ha dependències en el càlcul sobre un eix:
  - Files: quan les dependències són horitzontals.
  - Columnes: quan les dependències són verticals.

Per blocs:
- Cada processador processa segments de N*N/P elements de la matriu.
- Cada segment esta dividit en blocs de B columnes.
- Útil quan hi ha dependències de càlcul sobre els dos eixos.

Example 1: linear task decomposition

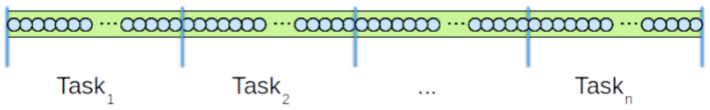
A task is a "code block" or a procedure invocation

```
int main() {
   ...
   tareador_start_task("init_A");
   initialize(A, N);
   tareador_end_task("init_A");

   tareador_start_task("init_B");
   initialize(B, N);
   tareador_end_task("init_B");

   tareador_start_task("dot_product");
   dot_product (N, A, B, &result);
   tareador_end_task("dot_product");
   ...
}
```

Example 2: iterative task decomposition

A task is a chunk of iterations of a loop, as for example, in the sum of two vectors

```
void vector_add(int *A, int *B, int *C, int n) {
   for (int i=0; i< n; i++) C[i] = A[i] + B[i];
}

void main() {
   ....
   vector_add(a, b, c, N);
   ...
}
```



Single loop iteration:

```
void vector_add(int *A, int *B, int *C, int n) {
   for (int i=0; i< n; ii++)
       tareador_start_task("singleit");
       C[i] = A[i] + B[i];
       tareador_end_task("singleit");
}
```

Chunk of loop iterations:

```
#define BS 16
void vector_add(int *A, int *B, int *C, int n) {
   for (int ii=0; ii< n; ii+=BS) {
       tareador_start_task("chunkit");
       for (i = ii; i < min(ii+BS, n), i++)
           C[i] = A[i] + B[i];
       tareador_end_task("chunkit");
       }
}
```

Iterative task decomposition (1)

Task granularity defined by the number of iterations out of the loop each task executes. For example, using **implicit tasks**:

```
void vector_add(int *A, int *B, int *C, int n) {
   int who =  omp_get_thread_num();
   int nt =  omp_get_num_threads();
   int BS = n / nt;
   for (int i = who*BS; i < (who+1)*BS; i++)
       C[i] = A[i] + B[i];
}

void main() {
   ....
   #pragma omp parallel
   vector_add(a, b, c, N);
   ...
}
```

Each implicit task executes a subset of iterations, based in the thread identifier executing the implicit task and the total number of implicit tasks (i.e., number of threads in the team).

Task granularity defined by the number of iterations each task executes. For example, using **explicit tasks**:

```
void vector_add(int *A, int *B, int *C, int n) {
   for (int i=0; i< n; i++)
       #pragma omp task
       C[i] = A[i] + B[i];
}

void main() {
   ....
   #pragma omp parallel
   #pragma omp single
   vector_add(a, b, c, N);
   ...
}
```

each explicit task executes a single iteration of the `i` loop, large task creation overhead, very fine granularity!

Iterative task decomposition (5)

- **Option 2:** `taskloop` construct to specify tasks out of loop iterations:

```
void vector_add(int *A, int *B, int *C, int n) {
   int BS = ...
   #pragma omp taskloop grainsize(BS)       // or alternatively num_tasks(n/BS)
   for (int i=0; i< n; i++)
       C[i] = A[i] + B[i];
   // Implicit task synchronization at the end of the taskloop due to the implicit taskgroup
}
void main() {
   #pragma omp parallel
   #pragma omp single
   ... vector_add(a, b, c, N); ...
}
```

  - `grainsize(m)`: each task executes $[min(m, n) \; .. \; 2 \times m)$ consecutive iterations, being n the total number of iterations
  - `num_tasks(m)`: creates as many tasks as $min(m, n)$

Example3: Non countable loops - list traversal example

List of elements, traversed using an uncountable (while) loop

```
int main() {
struct node *p;

p = init_list(n);
...

while (p != NULL) {
    tareador_start_task("computeNode");
    process_work(p);
    tareador_end_task("computeNode");
    p = p->next;
    }
...
}
```

List of elements, traversed using a `while` loop while not end of list

```
int main() {
   struct node *p;

   p = init_list(n);
   ...
   #pragma omp parallel
   #pragma omp single
   while (p != NULL) {
       #pragma omp task firstprivate(p) // see note below
       process_work(p);
       p = p->next;
       }
   ...
}
```

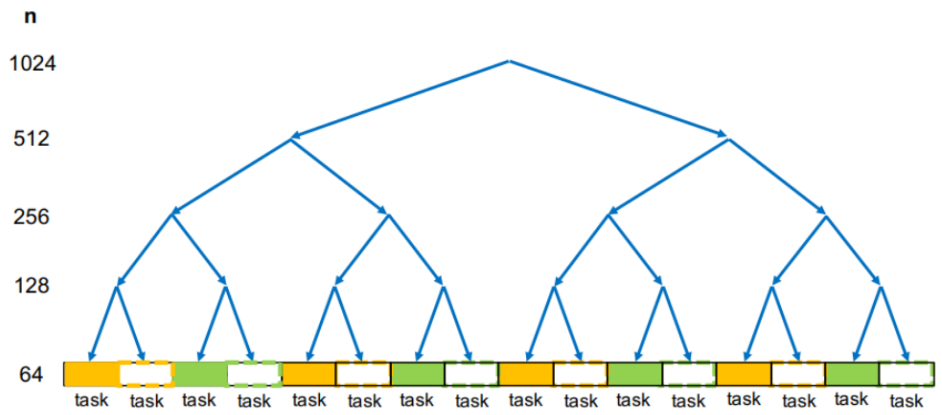
Granularity is one iteration, hopefully with sufficient work to amortise task creation overhead.

Note: `firstprivate` needed to capture the value of p at task creation time to allow its deferred execution.

Example 4: "Divide–and–conquer" task decomposition

Sum of two vectors by recursively dividing the problem into smaller sub-problems

- **Leaf strategy**: a task corresponds with each invocation of `vector_add` once the recursive invocations stop




```
#define N 1024
#define MIN_SIZE 64

void vector_add(int *A, int *B, int *C, int n) {
   for (int i=0; i< n; i++) C[i] = A[i] + B[i];
}

void rec_vector_add(int *A, int *B, int *C, int n) {
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       rec_vector_add(A, B, C, n2);
       rec_vector_add(A+n2, B+n2, C+n2, n-n2);
   }
   else
       {
       tareador_start_task("leaftask");
       vector_add(A, B, C, n);
       tareador_end_task("leaftask");
       }
}
void main() {
   ....
   rec_vector_add(a, b, c, N);
   ...
}
```

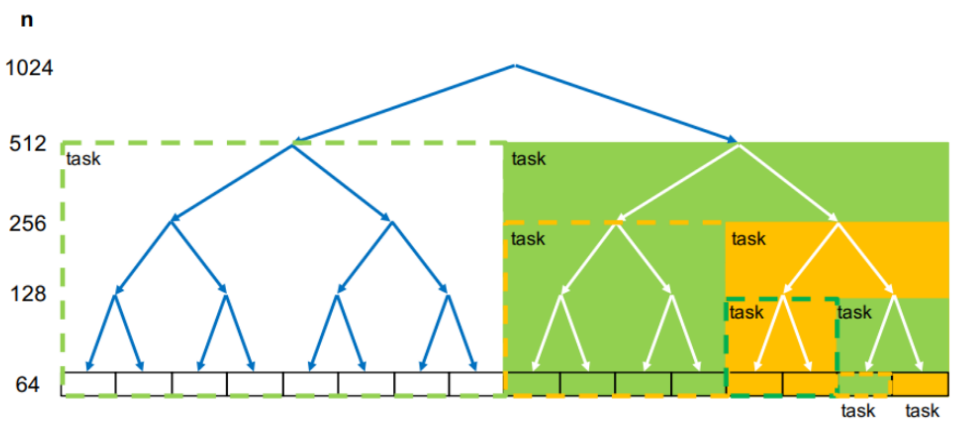
Recursive task decomposition: leaf strategy (2)

```
#define N 1024
#define MIN_SIZE 64
int result = 0;

  void dot_product(int *A, int *B, int n) {          {
     int tmp = 0;
     for (int i=0; i< n; i++)
        tmp += A[i] * B[i];
     #pragma omp atomic
     result += tmp;
  }

  void rec_dot_product(int *A, int *B, int n) {
     if (n>MIN_SIZE) {
         int n2 = n / 2;
         rec_dot_product(A, B, n2);
         rec_dot_product(A+n2, B+n2, n-n2);
     }
     else
         #pragma omp task
         dot_product(A, B, n);
  }

void main() {
   #pragma omp parallel
   #pragma omp single
   {
   rec_dot_product(a, b, N);
    #pragma omp takswait
   // Now we need the result here.
   ....
   }
}
```

Leaf strategy with **depth recursion control**

```
#define CUTOFF 2
...
void rec_dot_product(int *A, int *B, int n, int depth) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        if (depth == CUTOFF)
            #pragma omp task
            {
                rec_dot_product(A, B, n2 , depth+1);
                rec_dot_product(A+n2, B+n2, n-n2 , depth+1);
            }
        else {
            rec_dot_product(A, B, n2 , depth+1);
            rec_dot_product(A+n2, B+n2, n-n2 , depth+1);
        }

    }
    else    // if recursion finished, need to check if task has been generated
        if (depth <=  CUTOFF)
            #pragma omp task
            dot_product(A, B, n);
        else
            dot_product(A, B, n);
}
...
```

- **Tree strategy**: a task corresponds with each invocation of `rec_vector_add` during the *parallel* recursive execution




```
#define N 1024
#define MIN_SIZE 64

void vector_add(int *A, int *B, int *C, int n) {
   for (int i=0; i< n; i++) C[i] = A[i] + B[i];
}

void rec_vector_add(int *A, int *B, int *C, int n) {
   if (n>MIN_SIZE) {
        int n2 = n / 2;
        tareador_start_task("treetask1");
        rec_vector_add(A, B, C, n2);
        tareador_end_task("treetask1");
        tareador_start_task("treetask2");
        rec_vector_add(A+n2, B+n2, C+n2, n-n2);
        tareador_end_task("treetask2");
   }
   else vector_add(A, B, C, n);
}
void main() {
   ....
   rec_vector_add(a, b, c, N);
   ...
}
```

Recursive task decomposition: tree strategy (2)

```
int dot_product(int *A, int *B, int n) {
   int tmp = 0;
   for (int i=0; i< n; i++) tmp += A[i] * B[i];
   return(tmp);
}

int rec_dot_product(int *A, int *B, int n) {
   int tmp1, tmp2 = 0;
   if (n>MIN_SIZE) {
        int n2 = n / 2;
        #pragma omp task shared(tmp1) // firstprivate(A, B, n, n2) by default
        tmp1 = rec_dot_product(A, B, n2);
        #pragma omp task shared(tmp2) // firstprivate(A, B, n, n2) by default
        tmp2 = rec_dot_product(A+n2, B+n2, n-n2);
        #pragma omp taskwait
   } else tmp1 = dot_product(A, B, n);
   return(tmp1+tmp2);
}

void main() {
   #pragma omp parallel
   #pragma omp single
   result = rec_dot_product(a, b, N);
}
```

Tree strategy: where is the task synchronization? (3)

```
int result = 0;
void dot_product(int *A, int *B, int n);

void rec_dot_product(int *A, int *B, int n) {
   if (n>MIN_SIZE) {
        int n2 = n / 2;
        #pragma omp task
        rec_dot_product(A, B, n2);
        #pragma omp task
        rec_dot_product(A+n2, B+n2, n-n2);
   } else dot_product(A, B, n);
}

void main() {
   #pragma omp parallel
   #pragma omp single
   {
      #pragma omp taskgroup
     {
       rec_dot_product(a, b, N);
     }
     // Now we need the result here..
     ....
   }
}
```

Tree strategy with **depth recursion control**

```
#define N 1024
#define MIN_SIZE 64
#define CUTOFF 3

int rec_dot_product(int *A, int *B, int n, int depth) {
   int tmp1, tmp2 = 0;
   if (n>MIN_SIZE) {
        int n2 = n / 2;
        if (depth < CUTOFF) {
           #pragma omp task shared(tmp1)
           tmp1 = rec_dot_product(A, B, n2, depth+1);
           #pragma omp task shared(tmp2)
           tmp2 = rec_dot_product(A+n2, B+n2, n-n2, depth+1);
           #pragma omp taskwait
        } else {
           tmp1 = rec_dot_product(A, B, n2, depth+1);
           tmp2 = rec_dot_product(A+n2, B+n2, n-n2, depth+1);
        }
   }
   else tmp = dot_product(A, B, n);
   return(tmp1+tmp2);
}
```

```
#define N 1024
#define MIN_SIZE 64
#define CUTOFF 3

...

int rec_dot_product(int *A, int *B, int n, int depth) {
   int tmp1, tmp2 = 0;
   if (n>MIN_SIZE) {
      int n2 = n / 2;
      #pragma omp task shared(tmp1) final(depth >= CUTOFF) mergeable
      tmp1 = rec_dot_product(A, B, n2, depth+1);
      #pragma omp task shared(tmp2) final(depth >= CUTOFF) mergeable
      tmp2 = rec_dot_product(A+n2, B+n2, n-n2, depth+1);
      #pragma omp taskwait
   }
   else tmp1 = dot_product(A, B, n);
   return(tmp1+tmp2);
}
```

Llamadas y tipos de la librería:

```
//Número de threads de la región actual.
int omp_get_num_threads();
//Id del thread [0 - numThreads-1]
int omp_get_thread_num();
//Indica el número de threads a utilizar
void omp_set_num_threads(int n);
//Retorna el máximo número de threads
int omp_get_max_threads();
//Tiempo actual
double omp_get_wtime();
//Usar normalmente con:
double start = omp_get_wtime();
double end = omp_get_wtime();
printf("Work took %f seconds\n", end-start);

//Locks:
omp_lock_t lock;        //type of a lock
void omp_init_lock(&lock);    //initialize a lock
void omp_set_lock(&lock);
void omp_unset_lock(&lock);
int omp_test_lock(&lock);     //won't block the thread, 0 if fail to
acquire the lock
void omp_destroy_lock(&lock);
```

`#pragma omp parallel [Clausulas]`

- Crea una región paralela con los threads que podemos indicar con:
  - variable de entorno OMP_NUM_THREADS
  - Llamada a `void omp_set_num_threads(int n);`
  - Clausula num_threads()
- Tiene una barrera implícita al final.
- Clausulas
  - num_threads(N): Ignora el número de threads indicado fuera y la región paralela tendrá N threads.
  - if(exp_bool): Si la expresión evalúa falso, solo se creará un thread en la región.
  - shared(var-list): Todos los threads ven la misma variable
  - private(var-list): Todos los threads tienen una variable privada (**no inicializada!**).
  - firstprivate(var-list): Todos los threads tienen una variable privada inicializada.
  - reduction(<op>:var-list): Define cómo se efectuará (+, -, *, /, ^) el join cuando todos los threads acaben. Las variables de la lista se privatizan automáticamente.

`#pragma omp single [Clausulas]`

- La región de código sólo se ejecuta por un thread de la región paralela, los demás esperan.
- Tiene una barrera implícita al final
- Clausulas
  - private, firstprivate, shared
  - nowait : La barrera implícita desaparece y el resto de threads avanzan.

`#pragma omp barrier`

- Todos los threads de la región se esperan (sincronizan) en este punto. Una vez llegan todos siguen la ejecución.
- Las barreras implícitas de *single* i *parallell* son de este tipo.

`#pragma omp critical [(name)]`

- Soporte para la exclusión mútua. Solo habrá un thread a la vez en todas las regiones no nombradas.
- Podemos añadir un nombre a la zona de exclusión mutua para esa región en concreto.
- Los critical pueden sustituirse con los locks.

```
for(int i = 0;i < N; i++){
      if(i %2){
            #pragma omp critical(y)
            even++;
      }
      else{
            #pragma omp critical(x)
            odd++;
      }
}
```

`#pragma omp atomic [ update | read | write ]`

- Asegura lecturas, escrituras y actualizaciones son atómicas.
- Más eficiente que el critical usualmente.

`#pragma omp for [clausules]`

- Distribuye las iteraciones del siguiente bloque entre todos los threads de la región paralela. **El número de iteraciones tiene que ser conocido.**
- Clausulas
  - private, firsprivate, shared, reduction, nowait.
  - schedule[(type, [N])]: indica cómo distribuir las iteraciones
  - collapse(n): En bucles perfectamente anidados (sin nada enmedio y con el número de iteraciones definido en compilación), podemos juntar los N primeros loops.
  - ordered(n): Indica que hay que ordenar las iteraciones de los n bucles anidados. Para indicar las dependencias dentro del código, hay que usar:
    - depend(sink: expr) : Depende de la iteración resultante de expr (i-1).
    - depend(source) : Depende de toda la iteración actual.

private(expression)

- Quan en el constructor definim una variable private, es crea una variable nova amb el mateix nom a cada thread inicialment sense valor.
  No es necessària la sincronització.

firstprivate(expression)

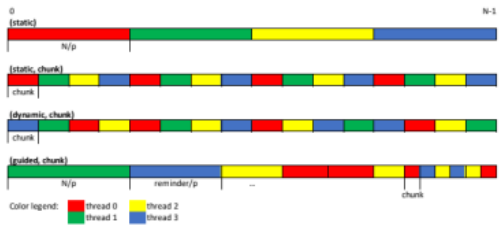
- El mateix que private però el valor de la variable esta inicialitzat amb el valor que tenia la variable inicialment.
  No es necessària la sincronització.

reduction(operator:list)

- Redueix l'acumulació de tots els threads en una variable.
  Els operadors poden ser: +,-,*,|,||.&,&&,^.
  Es crea una variable privada a cada thread i al final el compilador s'assegura que totes les acumulacions parcials es redueixen en la variable final (segons l'operador indicat).

Tipos de scheduling:

- **static:** Las iteraciones se dividen en bloques de tamaño N/num. threads. sha
- **static, N :** Las iteraciones se dividen en bloques de tamaño N.
- **dynamic[, N]:** Las iteraciones son "pedidas" por cada thread. Si no le añadimos la N, N = 1.
- **guided**: Variante de *dynamic* , el tamaño se reduce conforme al número restante.




```
#pragma omp task
{
      bloque codigo
}
```

- Se suele usar dentro de un *single*, normalmente solo queremos crear una tarea una vez.
- Clausulas:
  - shared, private, firstprivate
  - if(exp) Si exp == False, la tarea se crea y se ejecuta por el thread "que encuentra el pragma omp task".
  - final(exp) : Si exp == True, la tarea **y todas sus descendientes estarán marcadas como final** y se ejecutará secuencialmente por el thread que la crea (la estructura de la tarea se genera igualmente), se llama *included task*. Podemos ver si estamos en una tarea final con la llamada `bool omp_in_final()`.
  - mergeable : Aplicado a una tarea final, no crea la estructura de la tarea. (Realmente hace que se ejecute en el mismo espacio que la tarea que la crea).
  - depend : Marca las dependencias para las tareas. Las variables pueden estar dentro de un array.
    - depend(in : list-vars) : La tarea depende de un cálculo de la variable previa.
    - depend(out : list-vars) : El cálculo de las variables en esta tarea será dependiente para otra tarea.
    - depend(inout : list-vars) : Igual que depend(out)

Podemos tener puntos de sincronización en las tareas:

```
#pragma omp taskwait
```

Suspende el thread a la espera que acaben las tareas hijas de la actual tarea.

```
#pragma omp taskgroup
```

Suspende el thread a la espera que acaben todas las tareas descendientes de la actual tarea. En el *taskwait* solo T1, T2 y T4 están garantizados que acaben T1, T2 y T4. En el *taskgroup* T2, T3 y T4 están garantizados a acabar.

```
#pragma omp task {}      // T1
#pragma omp task         // T2
{
    #pragma omp task {}  // T3
}
#pragma omp task {}      // T4

#pragma omp taskwait
```

Ac

```
#pragma omp task {}          // T1
#pragma omp taskgroup
{
    #pragma omp task         // T2
    {
        #pragma omp task {}  // T3
    }
    #pragma omp task {}      // T4
}
```

Podemos controlar cuantas tareas generamos en un bucle:

```
#pragma omp taskloop [clausulas]
```

- Clausulas
  - shared, private, firstprivate, if, final, mergeable
  - grainsize(m):
  - num_tasks(n): Indica el número de tareas totales a generar.
  - collapse(n): Indica el número de loops que queremos juntar.
  - nogrpoup: Elimina el taskgroup implicit.

```
vector<int> myVector(N);
//First version -> using reduction
#ifdef V1
int sum=0;
int n=0;

#pragma omp parallel
#pragma omp for reduction(+:sum, n)
for(auto it: myVector){
      sum += it;
      n+=1;
}
#endif
```

```
//Second version -> shared/priv variables
#ifdef V2
int sumPriv = 0, nPriv = 0;
int sum = 0, n = 0;
#pragma omp parallel
{
      #pragma omp for firstprivate(nPriv, sumPriv)
      for(auto it: myVector){
          sumPriv += it;
          nPriv+=1;
      }
      #pragma omp critical(sum)
      sum += sumPriv;
      #pragma omp critical(n)
      n += nPriv;
}
#endif

cout << "Result: " << sum/n << endl;
```

LOCKS

```
void func(...){
      #pragma omp taskloop
      for(int i = 0; i < n; i++){
            índex = ...
            omp_set_lock(&x[index]);
            ...
            omp_unset_lock(&x[index]);
}     }
void main(){
      for(i = 0; i < ...; i++) omp_init_lock(&x[i]);
      #pragma omp parallel
      #pragma omp single
      func(...)
      for(i = 0; i < n; i++) omp_destroy_lock(&x[i]);
}
```