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.

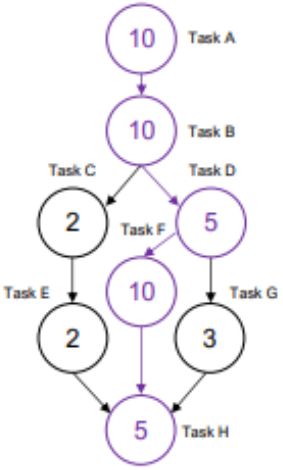
Live lock

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

Task dependence graph

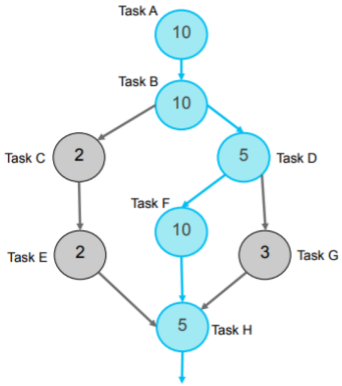
Graphical representation of the decomposition.

- 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.

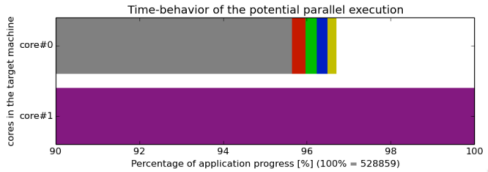
	Coarse grain	Fine grain	Medium grain
Number of tasks	1	n	n/m
Iterations per task	n	1	m
Parallelism	1	n	n/m

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

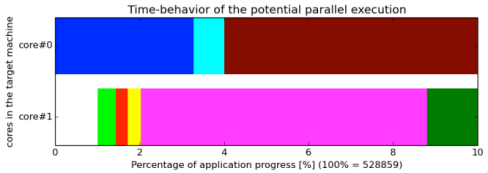
Coarse-grain decomposition: The whole loop is a task

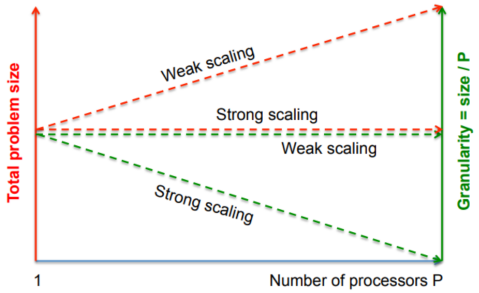
Fine-grain decomposition: Each iteration of the loop is a task

- Load unbalance



- Task creation overhead





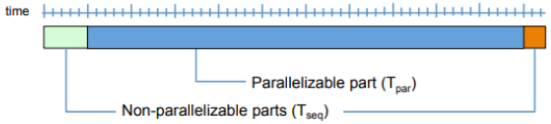
Strong vs. weak scalability

Two usual scenarios to evaluate the scalability of one application:

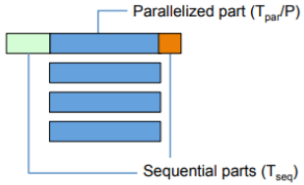
- Increase the number of processors P with constant problem size (strong scaling $\rightarrow$ reduce the execution time)
- Increase the number of processors P with problem size proportional to P (weak scaling $\rightarrow$ solve larger problem)

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

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

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

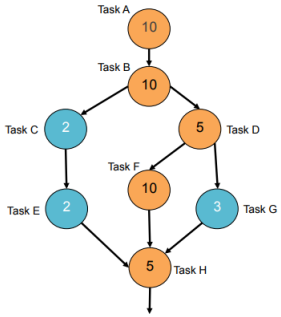
Two particular cases:

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

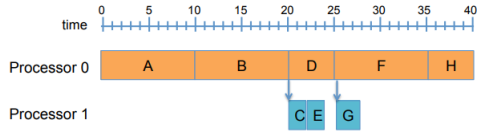
When $P \rightarrow \infty$ the expression of the speed–up becomes

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

Execution time bounds on P processors



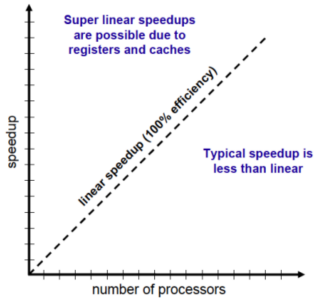
- T_p = execution time on P processors
- **Task scheduling**: how are tasks assigned to processors? For example:



- 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

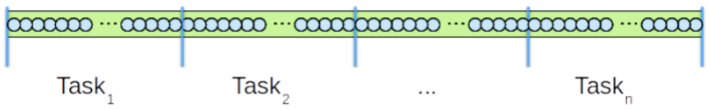
A task is a "code block" or a procedure invocation

```
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;
       }
   ...
}
```

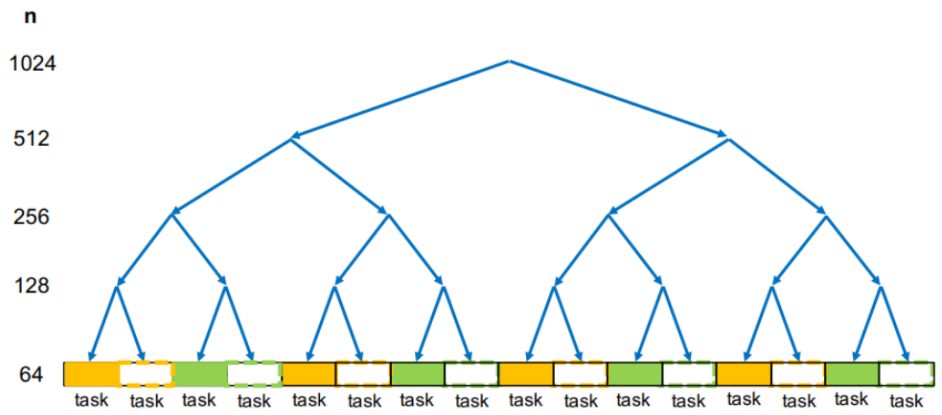
Granularity is one iteration, hopefully with sufficient work to amortise task creation overhead.

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);
   ...
}
```

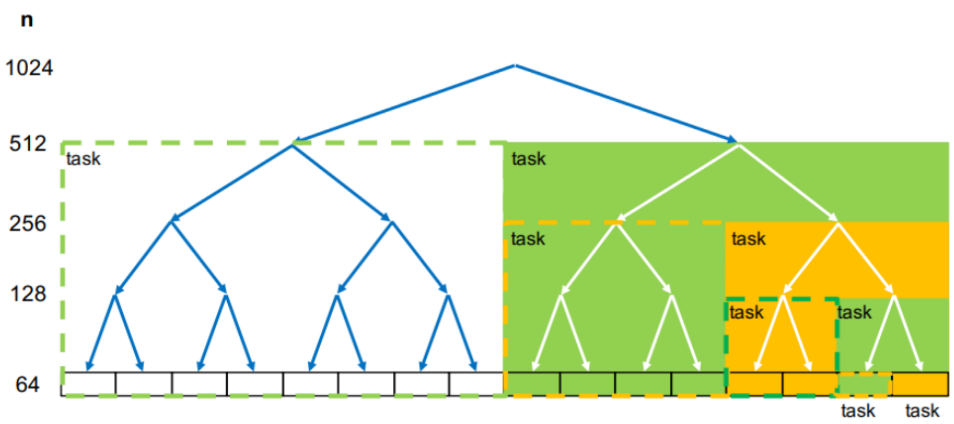
Recursive task decomposition: leaf strategy (2)

```
#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)

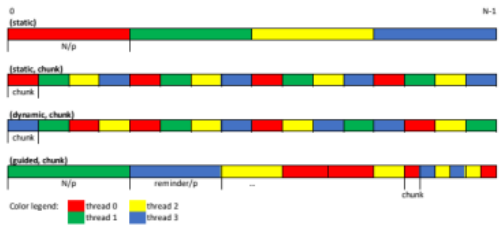
- El mateix que private però el valor de la variable esta inicialitzat amb el valor que tenia la variable inicialment.
  No es necessària la sincronització.

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]);
}
```

Sobre el uniprocesador podemos "conseguir" paralelismo de varias formas:

- Paralelismo a nivel de instrucción (procesadores superescalares) : Añadimos hardware extra en el pipeline y podemos ejecutar más instrucciones a la vez (en el mismo ciclo) DEL MISMO CONTEXTO DE EJECUCIÓN[2].

- Paralelismo a nivel de thread: "Rellenar" los ciclos vacíos (o perdidos por fallo de cache, latencia de memoria) del pipeline con instrucciones de diferentes contextos de ejecución.

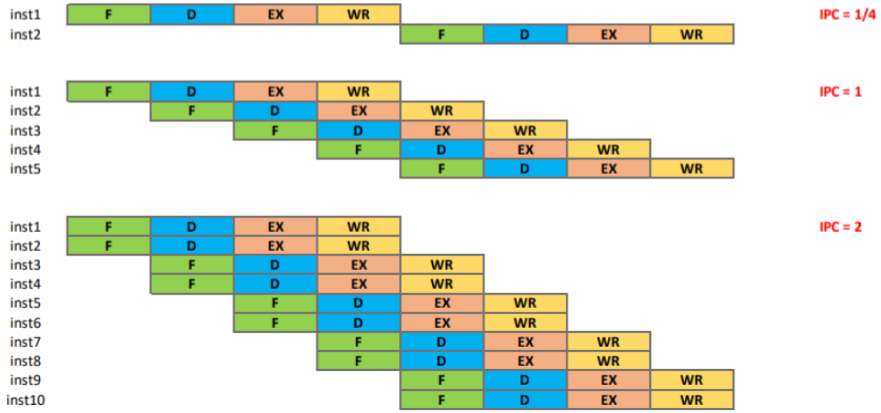
- Paralelismo a nivel de datos (instrucciones multimedia-SIMD[3]) : Se trata de calcular, una única instrucción sobre un gran conjunto de datos (por ejemplo los datos de un bucle for se pueden hacer todos en paralelo con una única instrucción SIMD).

En resumen, podemos conseguir paralelismo en un programa con:

- Principios de localidad espacial y temporal.
- Vectorizando operaciones con instrucciones SIMD.
- Alineando a bloque cache / memoria para un acceso más rápido
- Reordenación de instrucciones para explotar el paralelismo a nivel de instrucción.
- Típicos de todos los programas, a través de la jerarquía de memoria.
- Programación consciente de la arquitectura (PCA).

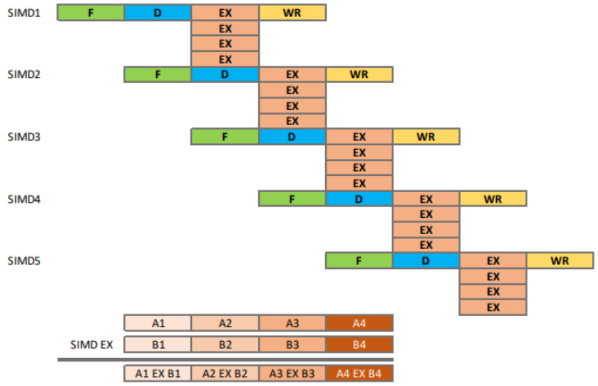
Pipelined and superscalar architecture

- Execution of single instruction divided in multiple stages
- Overlap the execution of different stages of consecutive instructions (pipelined) and consecutive instructions (superscalar)



SIMD architecture

- DLP (data-level parallelism): Single-Instruction executed on Multiple-Data (SIMD): vector functional unit



Memory hierarchy

- The principle of locality: if an item is referenced ...
  - Temporal locality: ... it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial locality: ... items whose addresses are close by tend to be referenced soon (e.g., straight line code, array access)
- Line (or block)
  - A number of consecutive words in memory (e.g. 32 bytes, equivalent to 4 words x 8 bytes)
  - Unit of information that is transferred between two levels in the hierarchy
- On an access to a level in the hierarchy
  - Hit: data appears in one of the lines in that level
  - Miss: data needs to be retrieved from a line in the next level

SIMD: Single Instruction Multiple Data: DLP (data-level parallelism)
Memory hierarchy:

- LOCALITAT:
  - Temporal: la dada es tornarà a accedir aviat (loop)
  - Espaial: accedir a les adreces del costat (vector, matriu)
- LÍNIA/BLOC: Paraules consecutives en memòria (32B = 4 words x 8B)
- ACCÉS:
  - Hit: la dada esa a una de les línies d'aquell nivell.
  - Miss: la dada s'ha de recuperar d'una linia del següent nivell.
- ALGORISMES DE REEMPLAÇAMENT: lru, fifo, lfu....
- WRITE POLICY: write-through, write-back (on hit), write-allocate, write-no-allocate (on miss)

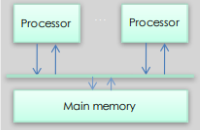
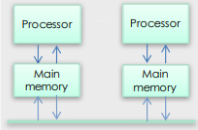
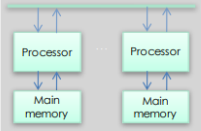
Elements of cache design

- Organization
  - Single vs. multilevel cache, Unified vs. split (instruction/data)
  - Cache size and line size
  - Addressing: logical vs. physical
- Placement algorithm
  - Direct, associative, set associative
- Replacement algorithm
  - Random, Least Recently Used (LRU), First-in First-out (FIFO), Least Frequently Used (LFU)
- Write (on hit) Policy
  - Write-through, Write-back
- Write (on miss) Policy
  - Write-allocate, write-no-allocate

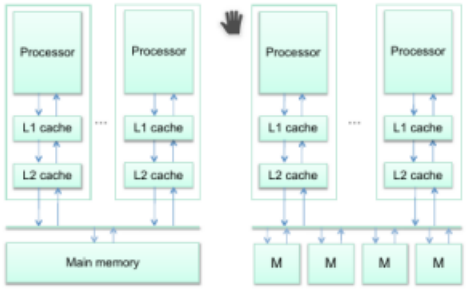
1. Organización de la jerarquía de memoria.
   a. Una única caché.
   b. Múltiples niveles de caché.
   c. Caché de datos y de instrucciones.

2. Política de asignación en la caché : ¿Dónde colocamos una dirección de memoria?
   a. Directa.
   b. Asociativa.
   c. Asociativa por conjuntos.
3. Política de reemplazo ¿Que hacer si no hay sitio y tenemos que traer un bloque?
   a. LRU
   b. FIFO
   c. etc...
4. Escritura en acierto : ¿Qué hacer si una escritura acierta en caché?
   a. Write-through : Actualizar MEM + CACHE.
   b. Write-back : Actualizar CACHE y memoria cuando la línea salga de la caché.
5. Escritura en fallo : ¿Qué hacer si escribimos y fallamos ?
   a. Write-allocate: Nos traemos la línea de caché.
   b. Write-no-allocate: No nos traemos la línea.

Classification of multi–processor architectures

Memory architecture	Address space(s)	Connection	Model for data sharing	Names
(Centralized) Shared-memory architecture	Single shared address space, uniform access time		Load/store instructions from processors	• SMP (Symmetric Multi-Processor) architecture • UMA (Uniform Memory Access) architecture
Distributed-memory architecture	Single shared address space, non-uniform access time		Load/store instructions from processors	• DSM (Distributed-Shared Memory architecture • NUMA (Non-Uniform Memory Access) architecture
	Multiple separate address spaces		Explicit messages through network interface card	• Message-passing multiprocessor • Cluster Architecture • Multicomputer

- Memoria centralizada (*shared-memory*): *Symmetric Multiprocessor Architecture* SMP
  - Un único espacio de direcciones y una única instancia de S.O.
  - El acceso a memoria de todos los procesadores es uniforme (U.M.A.).
  - El acceso a memoria se hace con instrucciones load/store.
  - **Existen problemas de coherencia de datos en memoria**



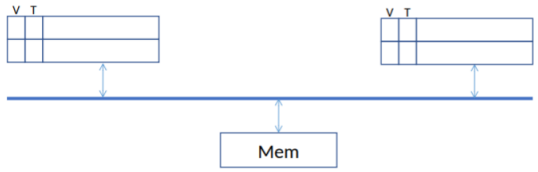
How to get coherence?

1) No caches (bad performance)
2) All cores share same L1 cache (bad performance)
3) Private write-through caches (Incoherent!)
4) Force read in one cache to see write made in another

   on the writer side:
   a) Broadcast writes to update other caches (write update coherence)
   b) Writes prevent hits to other copies (write-invalidate coherence)

   on the reader side:
   c) Writes broadcasted on shared bus (snooping)
   d) Each block assigned an ordering point (directory)

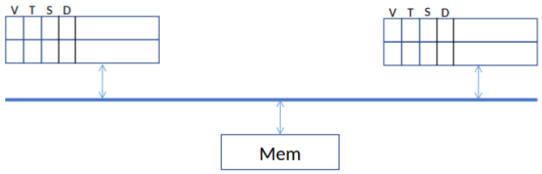
Write-update snooping coherence



Note 1: diagram showing the private caches of two processors and main memory. Each cache line has a valid bit (V), tag and data. Caches access the main memory through a bus.

Note 2: Listen to explanation in video for the specific example

Write-invalidate snooping coherence



Note 1: diagram showing the private caches of two processors and main memory. Each cache line has a valid bit (V), share bit (S) and dirty bit (D), tag and data. Caches access the main memory through a bus.

Note 2: Listen to explanation in video for the specific example

Coherence protocols:

- Write-update: writing processor broadcasts **the line** with the new value and forces all others to update their copies
- Write-invalidate: writing processor forces all others to invalidate their copies; **the line** with the new value is provided to others when requested or when flushed from cache

Coherence mechanisms:

- Broadcast-based (snooping): bus serves as broadcast mechanism to maintain coherency among copies of the same memory line in caches
- Directory-based: the sharing status of each line in memory is kept centralised in just one location (directory)

Write Update coherence: per cada escriptura s'ha d'enviar a totes les altres caches que poden tenir el valor i actualizar-lo

Write Invalidate coherence: després d'una escriptura a una de les cache, totes les cache que tenien una copia del valor s'han d'invalitzar i actuaran com si fos un miss el següent cop que vulguin accedir a la dada

Snooping: totes les escriptures són broadcast al bus compartit i l'ordre en el que apareixen en el bus és l'ordre en el que s'han de veure per tothom

Concepts in video lesson 5 (cont.)

The coherence problem:

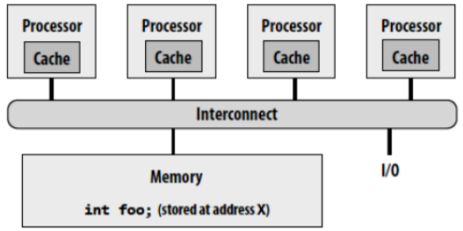


Chart shows value of `foo` (variable stored at address X) stored in main memory and in each processor's cache **

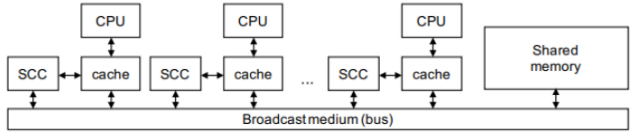
** Assumes write-back cache behavior

Action	P1 $	P2 $	P3 $	P4 $	mem[X]
					0
P1 load X	0 miss				0
P2 load X	0	0 miss			0
P1 store X	1	0			0
P3 load X	1	0	0 miss		0
P3 store X	1	0	2		0
P2 load X	1	0 hit	2		0
P1 load Y (say this load causes eviction of foo)		0	2		1

(CMU 15-418, Spring 2012)

Broadcast-based (snooping) coherence mechanism

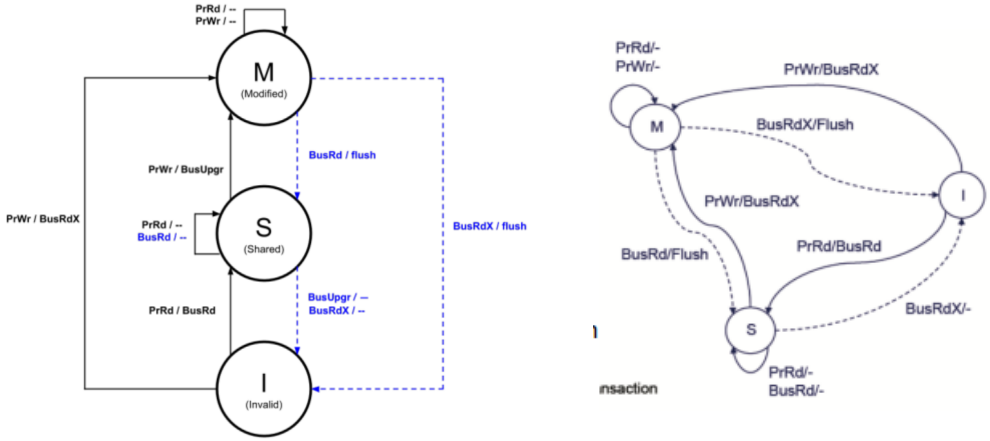
- Cache coherence is maintained at **cache line granularity**, NOT at the individual words inside the cache line
- Every cache that has a copy of a line from physical memory keeps its sharing status (status distributed)
- Broadcast medium (e.g. a bus) used to make all transactions visible to all caches and define **ordering**
- Caches monitor (snoop on) the medium and take action on relevant events (SCC: snoopy cache controllers)



Simple write-invalidate snooping protocol (MSI)

- A line in a cache memory can be in three different states:
  - Modified (M): dirty copy of the line
  - Shared (S): clean copy of the line
  - Invalid (I): invalidated copy of the line (not valid), or it does not exist in cache
- CPU events
  - PrRd (Processor read)
  - PrWr (Processor write)
- Bus events (caused by cache controllers)
  - BusRd: asks for copy with no intent to modify
  - BusRdX: asks for copy with intent to modify
  - BusUpgr: asks for permission to modify existing line, causes invalidation of other copies
  - Flush: puts line on bus, either because requested or voluntarily when dirty line in cache is replaced (WriteBack)

Simple write-invalidate snooping protocol (MSI)



- Who provides the line when requested via BusRd or BusRdX?
  - If line in S or I in other caches then main memory provides it
  - If line in M in another cache then this cache provides it (Flush)

Write-invalidate snooping protocol - MSI

En este protocolo, tendremos que cada CPU puede realizar lecturas y escrituras (eventos PrRd y PrWr) contra la jerarquía de memoria y el sistema de *broadcast* genera diferentes peticiones por el bus que ven todas las caches:

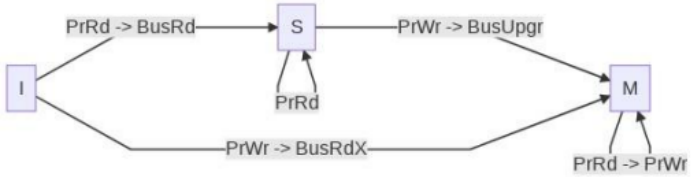
- BusRd: Petición de una línea para lectura.
- BusRdX: Petición de una línea con intención de modificarla.
- BusUpgr : Petición de escritura contra una línea de caché, causa la invalidación de las copias.
- Flush : Alguien nos pide el dato de caché o la política de reemplazo saca una linea en estado **M.**

Tenemos 3 posibles estados para una línea de caché:

1. **I**nvalido: La línea de caché no es válida o no existe en la caché.
2. **M**odificado: Copia "sucia" de la línea de caché.
3. **S**hared: Copia "limpia" y compartida de una línea de cache.

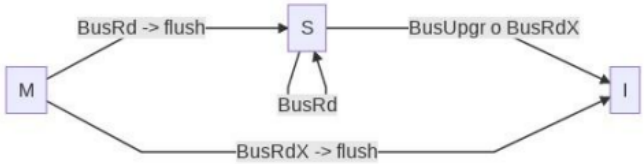
Tenemos dos diagramas de transición de estados:

1. En función de qué operación hace nuestra CPU y cómo afecta a nuestras líneas de caché.



En la imagen, -> se lee como "implica"

2. En función de qué operación realiza otra CPU y cómo afecta a nuestras líneas de caché. Cada caché observa los eventos del bus.



En la imagen, -> se lee como "implica". Si llega cualquier evento a I, nos quedamos y solo saldremos de él si necesitamos leer algo.

MSI optimizations. Thread–private lines

- MSI requires two bus transactions for the common case of read followed by write, both from the same processor (no sharing at all)
  - Transaction 1: BusRd to move from I to S state
  - Transaction 2: BusUpgr to move from S to M state
- **MESI** protocol adds E (Exclusive) clean state:
  - Cache line in E if only one clean copy of the line.
  - If write access by the same processor, the upgrade from E to M does not require a bus transaction (BusUpgr).
  - If line in E and another cache requests, it then cache line state changes from E to S.

Ejemplo de ejecución - MSI

Ejemplo de cómo varían los estados en una ejecución de instrucciones, con política de escritura retardada en acierto.

Evento	Cache P1	Cache P2	Cache P3	Status P1	Estado P2	Estado P3	MEM	Comentario
Init, T = 0	-	-	-	I	I	I	0	Si la linea no esta en caché, estado I
P1 ld X	0 (miss)	-	-	S	I	I	0	Estado I → Estado S a causa de una lectura (PrRd)
P2 ld X	0	0(miss)	-	S	S	I	0	IDEM anterior
P1 st 1, X	1	0	-	M	I	I	0	Estado S → Estado M Una PrWr del procesador invalida todas las demás copias.
P3 ld X	1	0	1 (miss)	S	I	S	0	P3 hace una lectura (de I → S) y el dato se lo proporcionará la caché de P1 (flush). El dato de P1 pasa a estar compartido (estado S)
P3 st 2, X	1	0	2	I	I	M	0	P3 tiene una copia válida, estado S → estado M e invalida las demás.
P2 load X	1	2	2	I	S	S	0	P2 tiene una copia invalida, P3 hace flush y proporciona la línea de cache a P2.
P1 load Y	mem[Y]	2	2	I	S	S	0	Flush no upd.
P2 load Y	mem[Y]	mem[Y]	2	I	I	S	2	Flush+ update

Resumen y escalabilidad

- El protocolo de coherencia actúa sobre las líneas de caché.
- Si estamos en los estados **I** o **S,** el dato lo proporciona la memoria principal.
- Solamente puede haber una única copia sucia en toda la caché.
- Si hay una copia sucia (estado **M**) en alguna caché y hay una petición sobre esta línea, esta caché proporcionará el dato.
- Hay variaciones/optimizaciones de este protocolo añadiendo más estados para controlar ciertos casos.
- Este tipo de protocolos de *broadcast* no escalan, la red se puede colapsar en algún momento por la cantidad de mensajes necesarios; podemos evitar esto teniendo un sistema centralizado (directorio).

True vs. false sharing

- True sharing
  - Data sharing is unavoidable in parallel computing. Coherence mechanisms are there to allow this data sharing; synchronization allows to share appropriately.
- False sharing
  - Cache line may also introduce artefacts: more that 1 (distinct) data object, or also multiple elements of same object, may reside in the same cache line
  - False sharing occurs when different processors make references (read and write) to those different objects or elements within the same cache line, thereby inducing "unnecessary" coherence operations.

True sharing example

Assume each task is executing an instance of the following `dot_product` function:

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

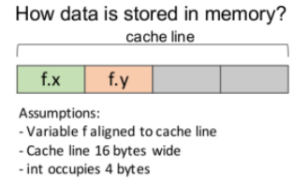
The line containing variable `result` is subject to coherence actions at each iteration of `i`. It could be easily transformed into

```
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;
```

to reduce cache coherence traffic. **Note:** `atomic` is used to guarantee exclusive access to variable `result`

False sharing example

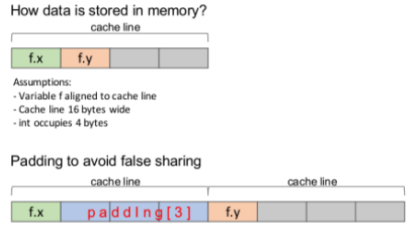
```
struct foo {
  int x, y; //x and y will reside in same cache line
} f; // aligned to cache line

void main() {
  int s=0;
  #pragma omp parallel
  #pragma omp single
  {
      #pragma omp task shared(s)
      for (int i=0;i<1000000;i++)
        s+=f.x

      #pragma omp task
      for (int i=0;i<1000000;i++)
        f.y++
  }
}
```



False sharing of the line containing fields x and y. How could we force fields x and y to be in different cache lines?

False sharing example (cont.)

```
struct foo {
  int x;
  int padding[3];
  int  y; //x and y will NOT reside in same cache line
} f; // aligned to cache line

void main() {
  int s=0;
  #pragma omp parallel
  #pragma omp single
  {
      #pragma omp task shared(s)
      for (int i=0;i<1000000;i++)
        s+=f.x

      #pragma omp task
      for (int i=0;i<1000000;i++)
        f.y++
  }
}
```



Add a dummy field in between to separate fields x and y in different cache lines: `int padding[PAD_SIZE];` How much padding?

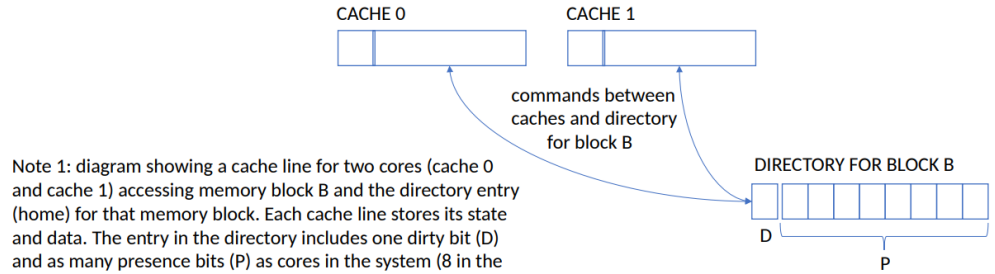
Bits per mantenir la coherència al sistema INSIDE numa node:
(#processadors *) [#linies * 2bits/linia, #linies = mida MC/ mida linia caché]

NUMA architectures (video lesson 5)
NUMA: Non-Uniform Memory Access
El temps d'accés és variable, depèn dels #salts que hagi de fer
Snooping: un sol bus on van totes les peticions dels cores. El bus es converteix en coll d'ampolla. No funciona bé amb més de 8-16 cores
Snoopy-invalidate: Quan escrius algo, invalides els següents perquè no es puguin llegir ni escriure.
Non-broadcast network: sobre la qual es van enviant les peticions.
- Distributed across cores: no centralitzat en un bus. No totes les peticions han d'anar a la mateixa part del Directory.

Bits per mantenir la coherència AMONG numa node:
(#nodes)* [(Mida MP/ Mida linia de MP) * (2+Presence bits/linia)]

Directory entry

- 1 dirty bit
- 1 bit/cache: present in that cache



Note 1: diagram showing a cache line for two cores (cache 0 and cache 1) accessing memory block B and the directory entry (home) for that memory block. Each cache line stores its state and data. The entry in the directory includes one dirty bit (D) and as many presence bits (P) as cores in the system (8 in the diagram)
Note 2: Listen to explanation in video for the specific example

Directory-based coherence

- Snooping:
  - Broadcast requests so others see them and to establish ordering
  - Bus becomes bottleneck!
  - Snooping does not work well with more than 8-16 cores

- Non-broadcast network
  - How do we observe requests we need to see?
  - How do we order requests to same block?

Directory

- Distributed across cores

- Each "slice" serves a set of blocks
  - One entry for each block it serves
  - Entry tracks which caches have block (in non-I state)
  - Order of accesses determined by "home" slice

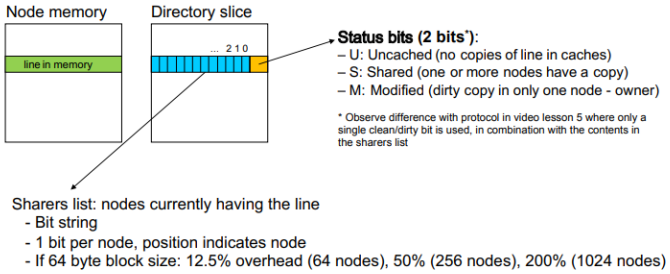
- Caches still have same states

Scaling of the broadcast mechanism

- Snooping schemes broadcast coherence messages to determine the state of a line in the other caches
  - Processor initiating access sends command to ALL other processors (having or not copy of the line)
  - Could be extended to support coherence in small NUMA systems, but does not scale to large number of nodes (excessive coherence traffic)
- Alternative: avoid broadcast by storing information about the status of each line in main memory, in the so called directory divided in slices, one slice per node
  - Each slice of the directory tracks the location of copies in caches of its memory lines
  - Coherence is maintained by point-to-point messages between the nodes

MSU directory-based cache coherency

- One slice of the directory associated to each node memory: one entry per line of memory
  - **Status bits**: they track the state of cache lines in its memory
  - **Sharers list**: tracks the list of remote nodes having a copy of a line. For small-scale systems, implemented as a bit string



- Directory slice is the "centralised" structure that "orders" the accesses to the lines in the associated node

Directory–based cache coherency (cont.)

- Who is involved in maintaining coherence of a memory line?
  - **Home** node: node where the line is allocated. It has the directory slice with the information to maintain its coherence.
  - **Local** node: node with the processor accessing the line
  - **Remote** nodes: **Owner** node containing **dirty** copy or **Reader** nodes containing **clean** copies of the line
- But ... how the **home** node for a memory line is decided?
  - **OS managed**, for example using a policy named *first touch*
    - The node that first "touches" a **page** will be the **home** node for all the lines in that page
    - For example, if memory pages are $P = 4$ KBytes and memory lines are $L = 128$ Bytes, then a page will contain $P/L = 32$ consecutive memory lines
    - Unless indicated differently we will assume that the number of memory lines in a page is 1 (i.e. $P = L$)

Simplified coherency protocol

Possible commands arriving to home node from local node:

- **RdReq**: asks for copy of line with no intent to modify
- **WrReq**: asks for copy of line with intent to modify
- **UpgrReq**: asks for permission to modify an existing line, invalidating all other copies

As a result of **RdReq** and **WrReq** the home node sends clean copy of line (**Dreply** command to local node). For **UpgrReq** it sends an acknowledgment (**Ack** command) to give permission.

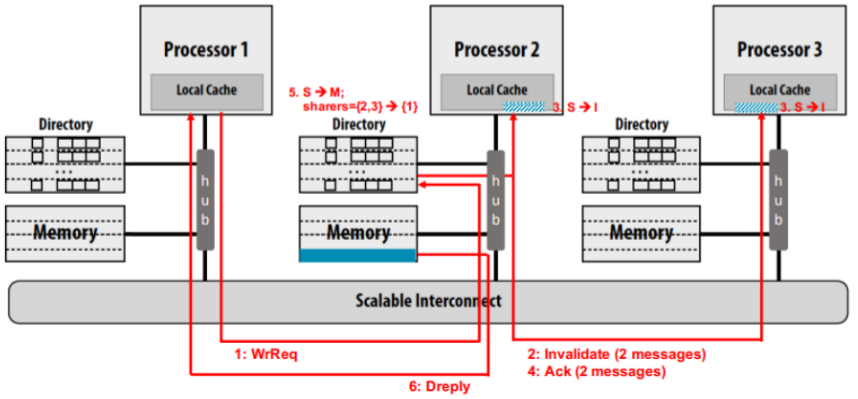
If needed the home node may generate other commands to remote nodes:

- **Fetch**: asks remote (owner) node for a copy of line (**Dreply**)
- **Invalidate**: asks remote (reader) node to invalidate its copy, remote sends confirmation to home (**Ack**)

Directory-based cache coherency: example

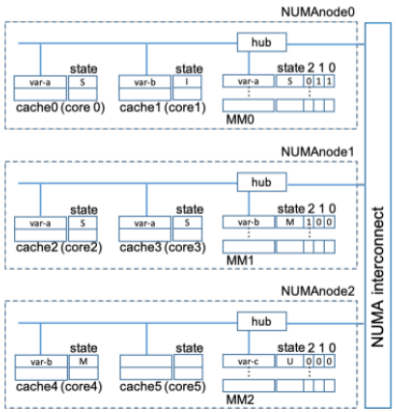
Write miss to clean line with two sharers

- Local node where the miss request originates: processor 1
- Home node where the memory line resides: processor 2
- Copies of line in caches of remote processors 2 and 3



Snooping- and directory-based protocols together!

If nodes have snoopy–based coherence, then the hub becomes an additional agent that interacts with the home (directory) nodes for the cache lines copied in the node



Coherence commands

- **Core:** $PrRd_i$ and $PrWr_i$, being I the core number doing the action
- **Snoopy:** $BusRd_j$, $BusRdX_j$ $BusUpgr_j$ and $Flush_j$, being j the snoopy/cache number doing the action
- **Hub/directoty:** $RdReq_{i\to j}$, $WrReq_{i\to j}$, $UpgrReq_{i\to j}$, $Dreply_{i\to j}$, $Fetch_{i\to j}$, $Invalidate_{i\to j}$, $Ack_{i\to j}$ and $WriteBack_{i\to j}$, from NUMAnode i to NUMAnode j

Line state in cache

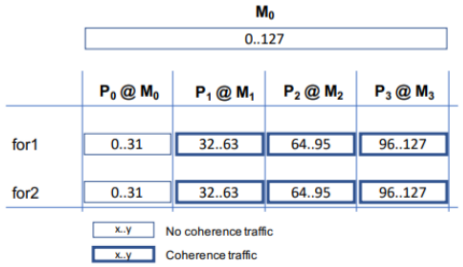
- M (Modified), S (Shared), I (Invalid)

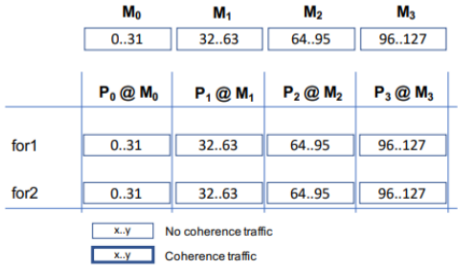
Line state in main memory

- M (Modified), S (Shared), U (Uncached)

Data sharing and initialization

```
#pragma omp parallel num_threads(4)
{
  int myid = omp_get_thread_num();
  int BS = 128 / omp_get_num_threads();
  for (int i=myid*BS; i<(myid+1)*BS; i++)
    b[i] = foo1(a[i]);
  for (int i=myid*BS; i<(myid+1)*BS; i++)
    a[i] = foo2(b[i]);
}
```



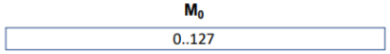


Data sharing and initialization

- True and false sharing have now a much higher penalty
- What may be wrong with data initialization? Be aware of "first touch"

```
for (int i=0; i<128; i++) {
    a[i] = random();
    b[i] = random();
}
```

Vectors a and b are allocated in a single node of of the NUMA system, as follows




```
#pragma omp parallel num_threads(4)
{
  int myid = omp_get_thread_num();
  int BS = 128 / omp_get_num_threads();
  for (int i=myid*BS; i<(myid+1)*BS; i++) {
    a[i] = random();
    b[i] = random();
  }
}
```

Vectors a and b are distributed across the memories of the NUMA system, as follows

How are synchronizations implemented?

```
#pragma omp atomic
var += non_protected_func();
```

```
#pragma omp critical
{
    // exclusive access
}
```

```
omp_set_lock(&lock_var);
// exclusive access
omp_unset_lock(&lock_var);
```

- In fact, entry to and exit from `critical` is the same as `omp_set_lock` and `omp_unset_lock`, respectively, but using an implicit (hidden) `omp_lock_t` variable
- `atomic` could also be implemented with `omp_lock_t`, but usually there is much better support at the architecture level (see later ...)
- `test-and-set`: read value in location and set to 1
  Example: test-and-set based lock implementation

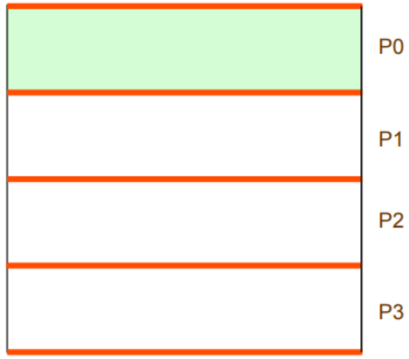
```
set_lock:   t&s r2, flag
            bnez r2, set_lock     // already locked?
            ...
unset_lock: st flag, #0           // free lock
```

INPUT BLOCK GEOMETRIC DATA DECOMPOSITION:
Requereix sincronització en l'update del output del vector del qual s'estigui fent la DD. Per reduir aquesta sync a 1 (només al final), s'ha d'usar una variable temporal tmp on anar acomulant els resultats dins dels inner loops: **scalar replacement**.

- Declarar com a private les variables que calgui // **#pragma omp parallel private(var)**
- Variables necessaries:
  - id = omp_get_thread_num();
  - nt = omp_get_num_threads();
  - bs = C/nt;
  - reminder = C%nt;
  - extra = id < reminder, extraprev = extra? id : reminder;
  - lowerb = id*bs + extraprev
  - upperb = lowerb + bs + extra
- for d'accés normal, i = 0, i < R ++i // j= loweb, j < upperb, ++j i abans d'actualitzar vector **#pragma omp atomic**

Geometric Block data decomposition INPUT

```
omp_lock_t Locks[256];

for (i = 0; i < 256; i++)
    omp_init_lock(&Locks[i]);

#pragma omp parallel private (index, i)
{
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    int BS = n / howmany;
    int rest = n % howmany;
    int start = myid * BS + ((rest > myid) ? myid : rest);
    int end = myid * BS + (rest > myid);

    for (i = start; i < min(end, n); i++) {
        index = ...;
        omp_set_lock(&Locks[i]);
        ...
        omp_set_lock(&Locks[i]);
    }
}

for (i = 0; i < 256; i++)
    omp_destroy_lock(&Locks[i]);
```

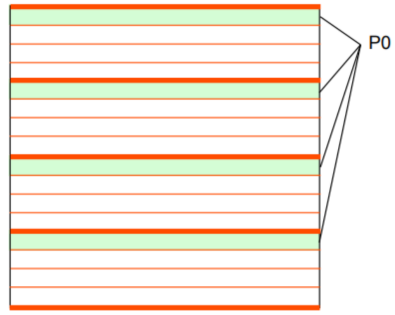


Geometric Block data decomposition OUTPUT

```
#pragma omp parallel private (index, i)
{
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    +int BS = n / howmany;
    int rest = n % howmany;
    int start = myid * BS + ((rest > myid) ? myid : rest);
    int end = myid * BS + (rest > myid);

    for (i = 0; i < n; i++) {
        index = ...;
        if (index >= start && index < end)
            ...
    }
}
```

Geometric Cyclic data decomposition INPUT

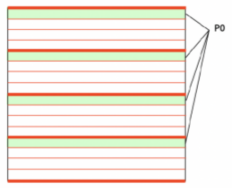
```
omp_lock_t Locks[256];

for (i = 0; i < 256; i++)
    omp_init_lock(&Locks[i]);

#pragma omp parallel private (index, i)
{
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for (i = myid; i < n; i += howmany) {
        index = ...;
        omp_set_lock(&Locks[i]);
        ...
        omp_set_lock(&Locks[i]);
    }
}
```



Geometric Cyclic data decomposition OUTPUT

```
#pragma omp parallel private (index, i)
{
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for (i = 0; i < n; i++) {
        index = ...;
        if (index % howmany == myid)
            ...
    }
}
```

CYCLIC DATA DECOMPOSITION, by ROWS

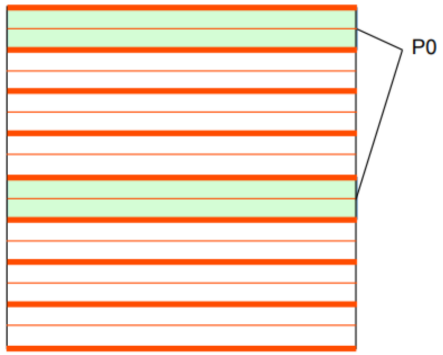



```
#pragma omp parallel private (i, j)
{
    int my_id = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for (int i=my_id; i<N; i+= howmany)
        for (int j=0; j<N; j++)
            ... m[i][j] ... // Input and/or Output
    ...
}
```

FUNCIO MIN
#define min(a,b) ((a) > (b) ? (b) : (a))

Geometric Block-Cyclic data decomposition INPUT

```
omp_lock_t Locks[256];

for (i = 0; i < 256; i++)
    omp_init_lock(&Locks[i]);

#pragma omp parallel private (index, i)
{
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    int BS = n / howmany;
    int rest = n % howmany;
    int start = myid * BS;
   int step = howmany * BS;

   for (int ii = start; ii < n; ii += step) {
       for (i = ii; i < min(ii + BS, n); i++) {
           index = ...;
           omp_set_lock(&Locks[i]);
           ...
           omp_set_lock(&Locks[i]);
       }
   }
}
```



Geometric Block-Cyclic data decomposition OUTPUT

```
#pragma omp parallel private (index, i)
{
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for (i = 0; i < n; i++) {
        index = ...;
        if ((index/BS) % howmany == myid)
            ...
    }
}
```

OUTPUT BLOCK-CYCLIC GEOMETRIC DATA DECOMPOSITION:

- Definir número d'elements per linia de cache
- #pramgma omp parallel private(variables)
- Variables que necessitem: id, nt, BS = num_elements_per_cache_line, loweb = id*BS, step = nt*BS
- For per fer salts de blocs (ii = loweb, ii < R; ii+=step)
    - For per navegar per cada bloc (i = ii, i < min(R, ii+BS), i++)
        - codi del for normal amb scalar replacement

2D Block / Block data decomposition

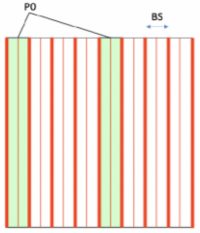
```
#pragma omp parallel private (i, j)
{
   int my_i = omp_get_thread_num()/4;
   int my_j = omp_get_num_threads()%4;
   int BSi = N/2;
   int BSj = N/4;
   int i_start = my_i * BSi;
   int i_end = i_start + BSi;
   int j_start = my_j * BSj;
   int j_end = j_start + BSj;
   for (int i=i_start; i<i_end; i++)
      for (int j=j_start; j<j_end; j++)
   ...
}
```

Example: matrix multiply using implicit tasks (3)

Let's reduce the load unbalance to 1 iteration at most ...

```
void matmul (double C[MATSIZE][MATSIZE],
             double A[MATSIZE][MATSIZE],
             double B[MATSIZE][MATSIZE]) {
int i, j, k;

#pragma omp parallel
   {
   int myid = omp_get_thread_num();
   int howmany = omp_get_num_threads();
   int i_start =  myid * (MATSIZE/howmany);
   int i_end = i_start + (MATSIZE/howmany);
   int rem = MATSIZE % howmany;
   if (rem != 0) {
       if (myid < rem) {
           i_start += myid;
           i_end += (myid+1);
       } else {
           i_start += rem;
           i_end += rem;
       }
   }
   ...
   }
}
```

BLOCK-CYCLIC DATA DECOMPOSITION, by COLUMNS




```
#pragma omp parallel private (i, j)
{
    int my_id = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for (int i=0; i<N; i++)
        for (int jj=my_id*BS; jj<N; jj+=howmany*BS)
            for (int j=jj; j<jj+BS; j++)
                ... m[i][j] ... // Input
                                // and/or Output
        ...
}
```