Paralelismo (PAR)

Cuatrimestre 4

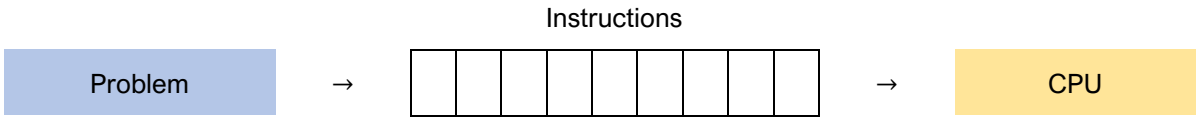
https://github.com/AdriCri22/Paralelismo-PAR-FIB

Unit 1: Why parallel computing?

A program is composed of a sequence of instructions (arithmetic, memory read and write, control…)

Serial execution

Instructions are executed one after another, only one at any moment in time.



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

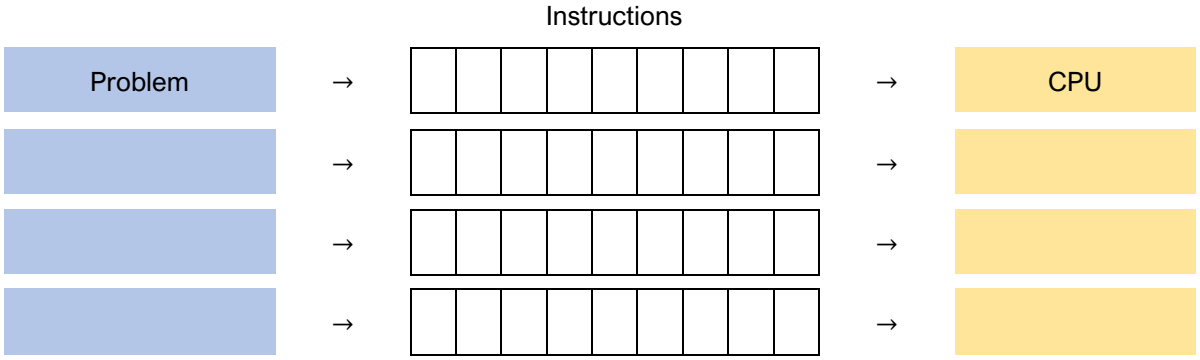
$$T = N/F$$

Ways to improve execution time:

- Optimizing the program to decrease the number of instructions.
- Increasing F, the number of executions per second.
- With parallel execution

Parallel execution

Consist of split the program into discrete parts, called tasks, and use multiple processors (CPUs), to execute them at same time.



To manage data, we have two options:

- Shared memory: offers a single memory space used by all processors. Processors do not have to be aware where data resides, except that there may be performance penalties, and that race conditions are to be avoided
- Distributed memory: computational tasks can only operate on local data, and if remote data are required, the computational task must communicate with one or more remote processors

Ideally, each processor could receive $1/P$ of the program, reducing its time execution time by P

$$T = (N/P)/F$$

Need to manage and coordinate the execution of tasks, ensuring correct access to shared resources, all extra time of data management its not included in this ideally execution time.

Throughput

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

Potential problems

- **Data race**: multiple tasks read and write some data and the final result depends on the relative timing of their execution.
- **Deadlock**: two or more tasks are unable to proceed because each one is waiting for one of the others to do something.
- **Starvation**: a task is unable to gain access to a shared resource and is unable to make progress.
- **Livelock**: two or more tasks continuously change their state in response to changes in the other tasks without doing any useful work.

Processors vs processes vs threads

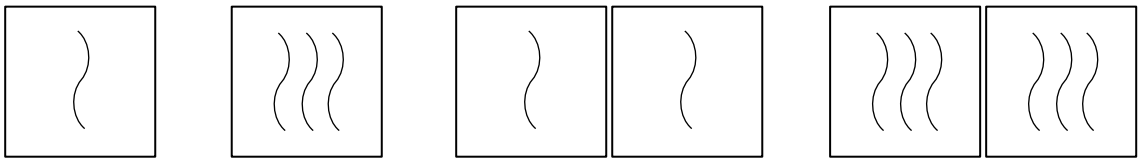
Processors are the hardware units that physically execute those logical computing agents, in most cases, there is a one-to-one correspondence between processes/threads and processors, but not necessarily (it is a OS decision).

Processes:

- Instance of a program that is being executed.
- Independent of each other.
- Don't share memory or other resources.
- Can create other processes to perform multiple tasks at a time (these are known as clone or child process, the main one is called parent process)

Threads:

- Subset of the process, also known as lightweight process (A process can have more than one thread).
- Are managed independently by a scheduler.
- All threads of one process are interrelated to each other.
- Share memory (data segment, code segment, files, etc.).
- Have their own registers, stack and counter.



One proccess
One thread

One process
Multiple threads

Multiple processes
One thread per process

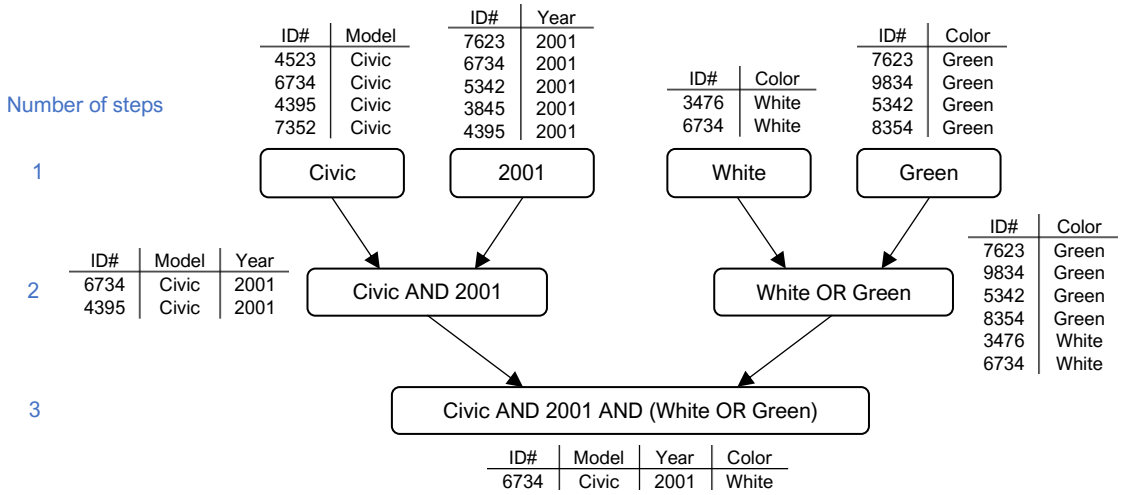
Multiple processes
Multiple thread per process

Unit 2 – Understanding Parallelism

Let's consider a database, if we want to obtain some data, we traverse all the records in the database and check the conditions we set. An example of this could be:

```
MODEL = ‘‘CIVIC’’ AND YEAR = 2001 AND (COLOR = ‘‘GREEN’’ OR COLOR = ‘‘WHITE)
```

A possible query plan could be:



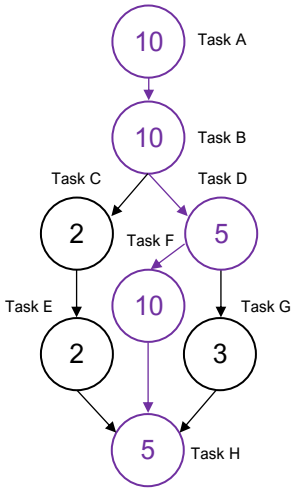
Each of these operations in the quey plan could be a task, each computing an intermediate table of entries that satisfy particular conditions.

- Some of them are independent. For example: "Civic", "2001", "Green" and "White".
- Others are not independent. For example: "Civic AND 2001", can't start its execution until both tasks "Civic" and "2001" complete.

Dependencies impose task execution ordering constrains that need to be fulfilled in order to guarantee correct results.

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*.

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

Unit 3: Introduction to parallel architectures

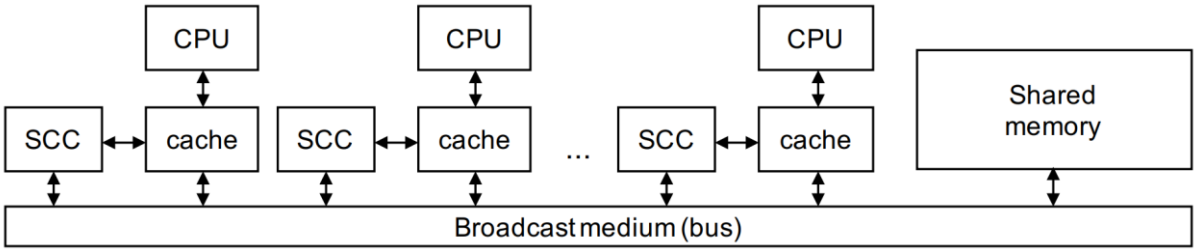
UMA (Uniform Memory Access time) || SMP (Symmetric) multiprocessors

Coherence protocols

1. Write-update: writing processor broadcasts the line with the new value and forces all others to update their copies.
2. 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.

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)



Protocol

A line in a cache memory can be in three different states:

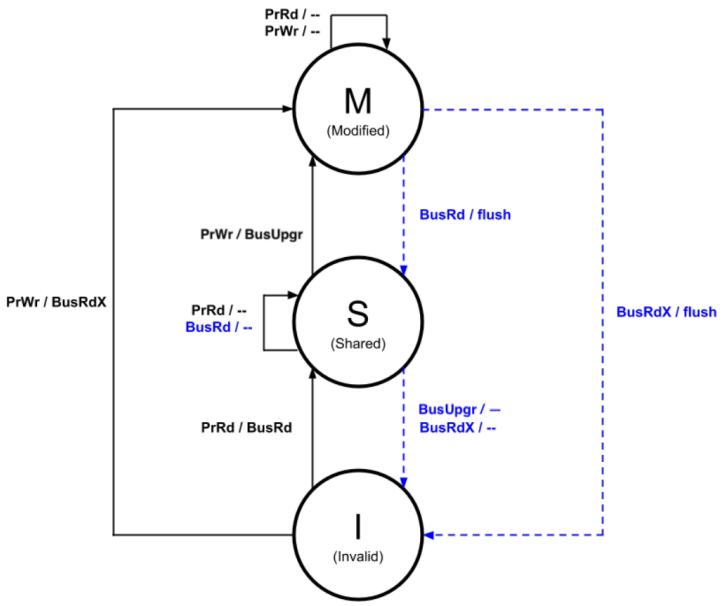
- Modified (M): dirty copy of the line
- Shared (S): clean copy of the line
- Invalid (I): invalidated copy of the line (not valid), or it does not exist in cache

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)



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)

MSI optimizations

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.

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.

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;
```

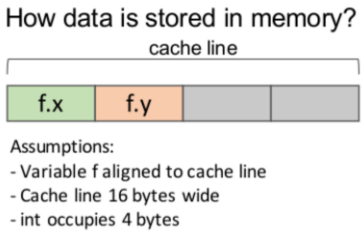
```
}
```

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

False sharing

- 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.

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




```
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++
   }
}
```

NUMA (Non-Uniform Memory Access time)

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)

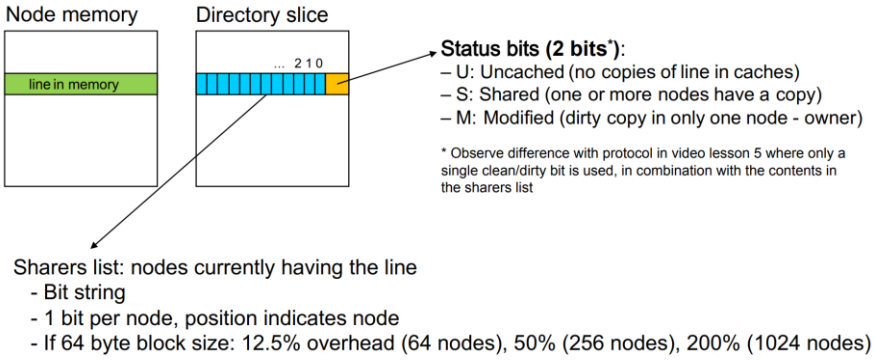
Alternative: avoid broadcast by storing information about the status of each line in main memory, in the so called directory divided in slices, one slice per node

- 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 (OS managed, for example first touch). 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

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

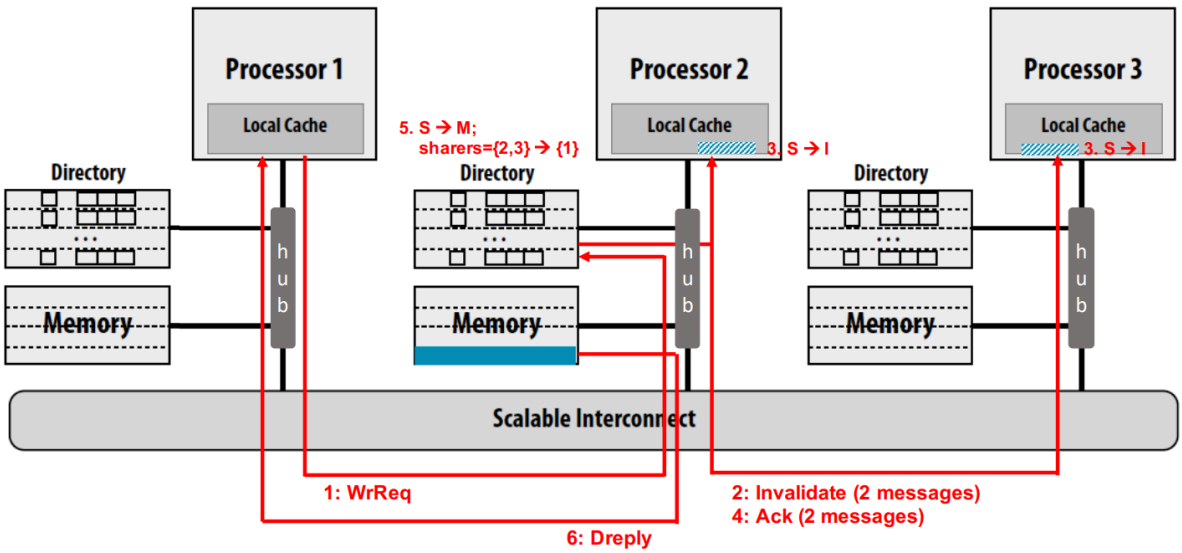
As a result of RdReq and WrReq the home node sends clean copy of line (Dreply command to local node). For UpgrReq it sends an acknowledgment (Ack command) to give permission. If needed the home node may generate other commands to remote nodes:

- 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

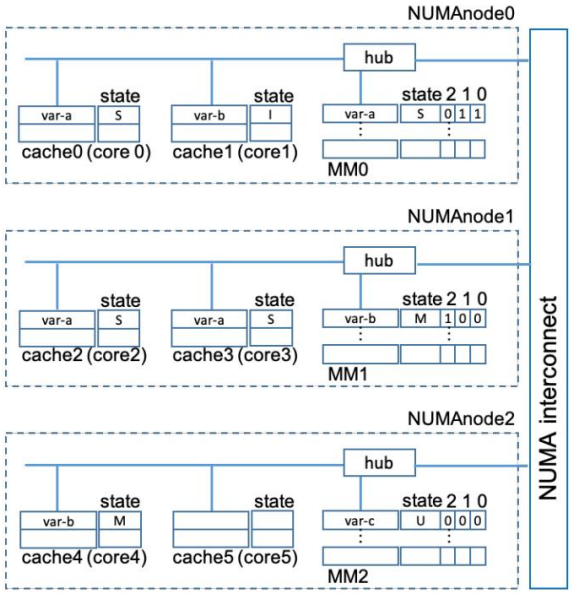
Write miss to clean line with two sharers

- 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)

Unit 4: Task decompositions

Iterative task decomposition

In iterative task decomposition strategies one can control task granularity by setting the number of iterations executed by each task.

Implicit tasks

```
#omp pragma parallel
{
    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 = start + BS + (rest > myid);

    for (i = start; i < end; i++)
        ...
}
```

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).

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!

Granularity: chunk of BS loop iterations

- **Option 1**: requires loop transformation

```
// Variation 1
#pragma omp parallel
#pragma omp single
for (i = 0; i < n; i++)
    #pragma omp task
    ...

// Variation 2
#pragma omp parallel
#pragma omp single
for (i = 0; i < rows; i++)
    #pragma omp task;
    for (j = 0; j < columns; j++)
        ...

// Variation 3
#pragma omp parallel
#pragma omp single
```

```
#pragma omp task
{
    int tmp = 0;
    for (int i = 0; i < n; i++)
        tmp += ...;
    #pragma omp atomic
    res += tmp;
}
```

Outer loop jumps over chunks of BS iterations, inner loop traverses each chunk

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

```
// Variation 1
#pragma omp parallel
#pragma omp single
#pragma omp taskloop grainsize(BS)
for (i = 0; i < n; i++)
    ...

// Variation 2
#pragma omp parallel
#pragma omp single
#pragma omp taskloop grainsize(BS) reduction(+: res)
for (i = 0; i < n; i++)
    res += ...;

// Variation 3
#pragma omp parallel
#pragma omp single
#pragma omp taskloop grainsize(BS) private(j)
for (i = 0; i < rows; i++) {
    int tmp = 0;
    for (j = 0; j < columns; j++)
        tmp += ...;
    res += tmp;
}
```

Uncountable loop

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.

Recursive task decomposition

Leaf strategy

```
#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);
}
```

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

```
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);
}
```

With **depth recursion control**:

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

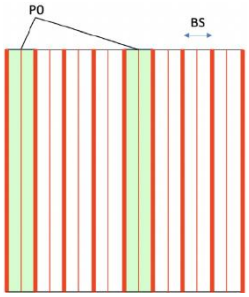
```
...
```

```
int rec_dot_product(int *A, int *B, int n, int depth) {
   int tmp1, tmp2 = 0;
   if (n>MIN_SIZE) {
      int n2 = n / 2;
      #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);
}
```

…

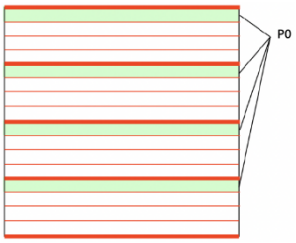
Unit 5: Data-aware task decomposition strategies

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 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]);
    }
}

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

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]);
        }
    }
}

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 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)
            ...
    }
}
```

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)
            ...
    }
}
```

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++)
   ...
}
```