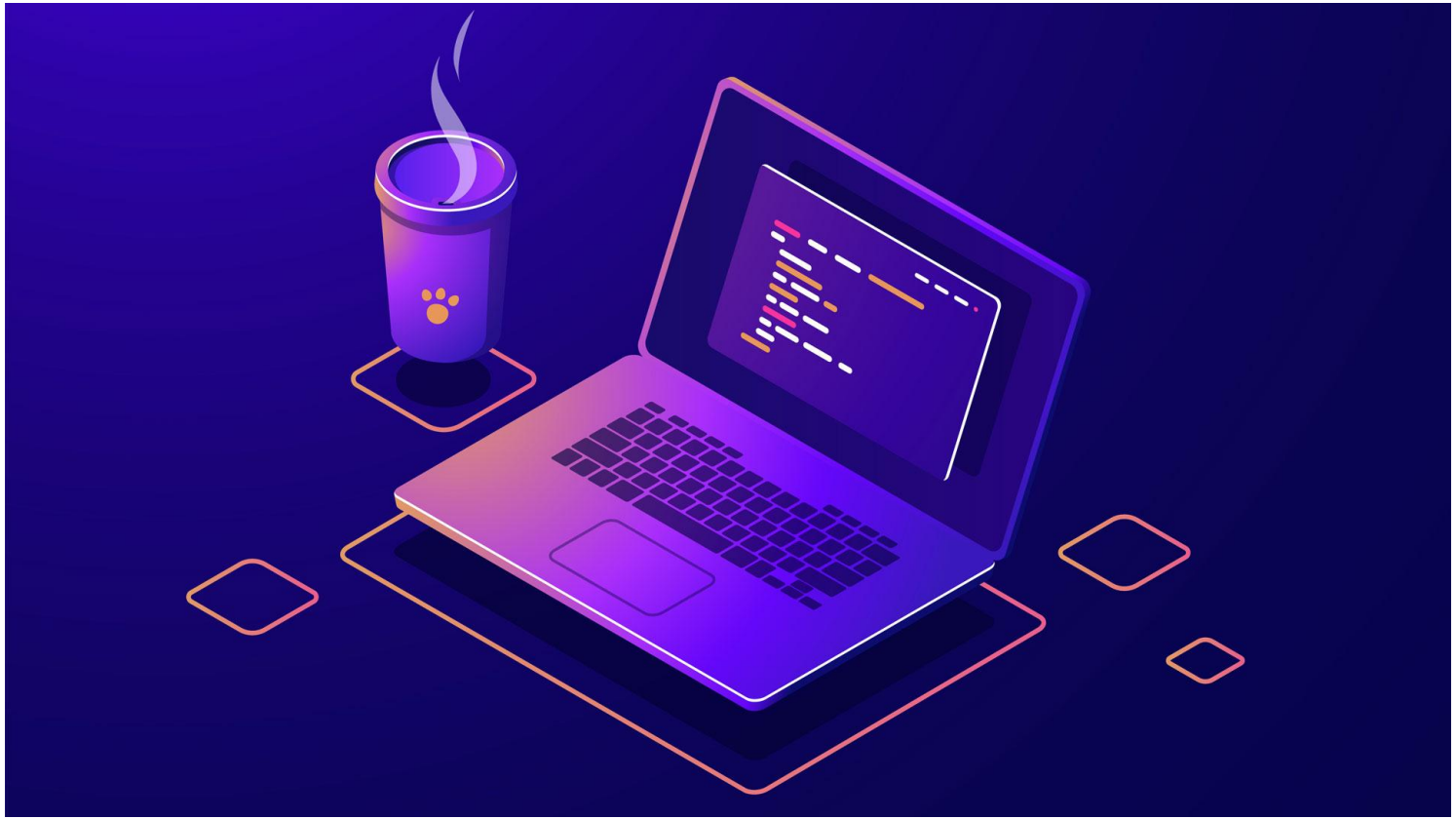
Task Generation Control



Task Generation Control

- Iterative Task Decompositions
- Recursive Task Decompositions

Task Generation Control

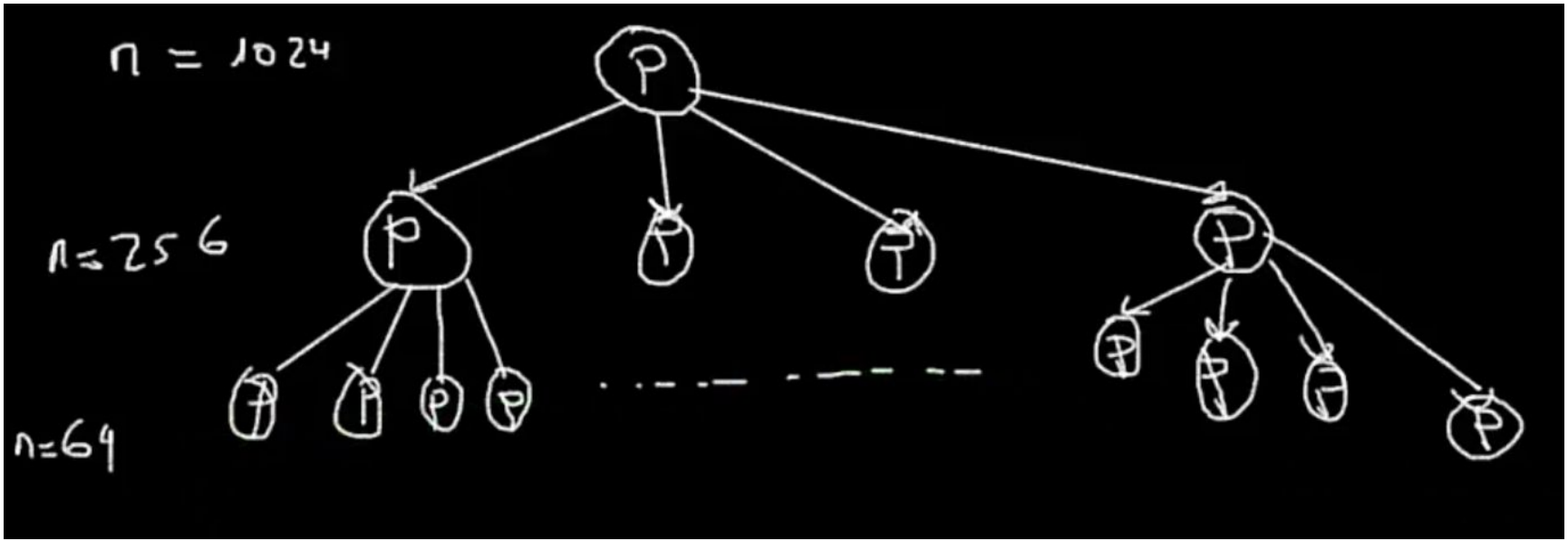
- Iterative Task Decompositions
- Recursive Task Decompositions

Task Generation Control

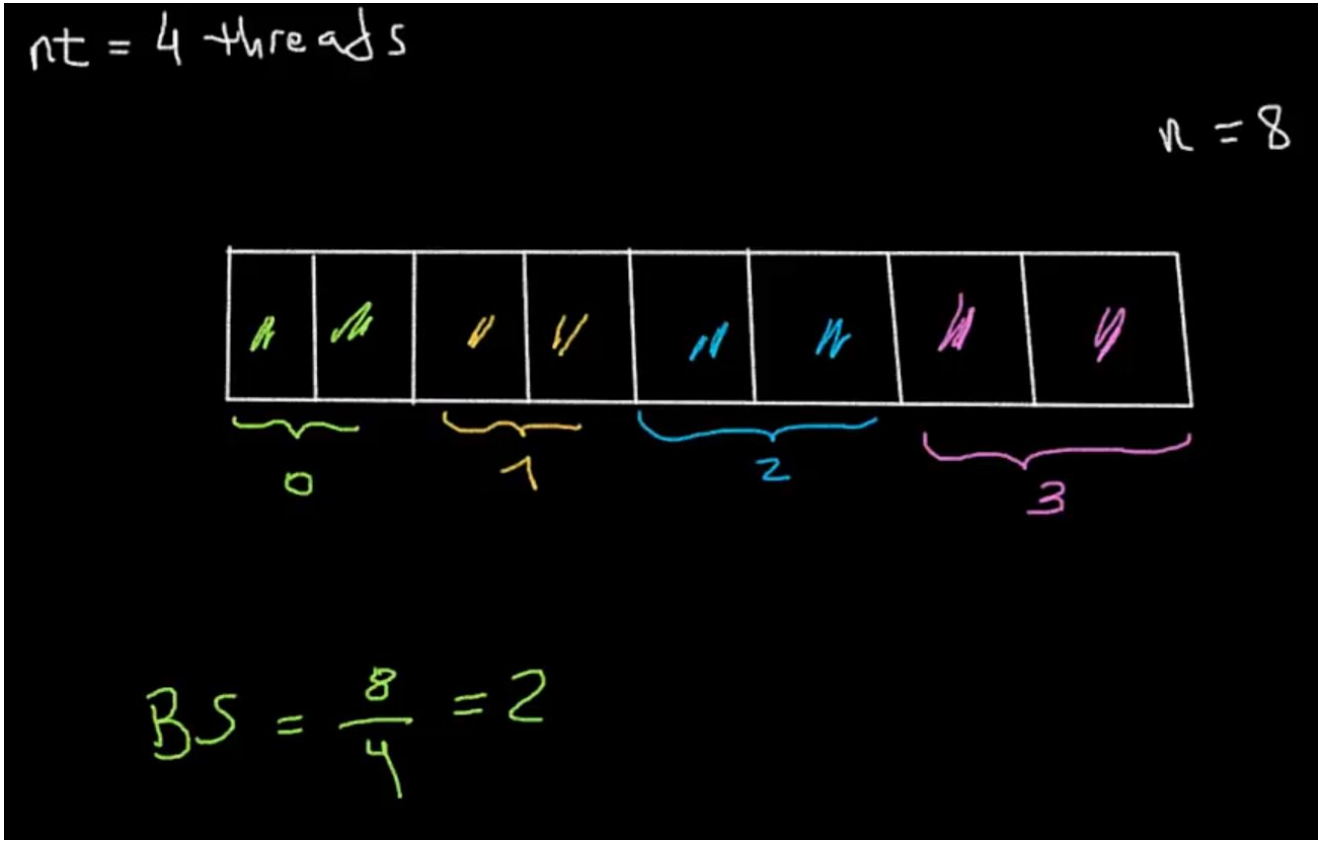
Excessive task generation may not be necessary (i.e. cause excessive overhead): need mechanisms to control number of tasks and/or their granularity

- In iterative task decomposition strategies one can control task granularity by setting the number of iterations executed by each task
- In recursive `task` decomposition strategies one can control task granularity by controlling recursion levels where tasks are generated (**cut-off control**)
  - after certain number of recursive calls (static control)
  - when the size of the vector is too small (static control)
  - when there are sufficient tasks pending to be executed (dynamic control)

Example of Cut Off (We will study deeply later)



Another Example on the size of vector



Task Generation Control

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 Generation Control

Iterative task decomposition (2)

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!

Task Generation Control

How do we make Explicit Tasks execute a chunk of iterations?

Task Generation Control

Iterative task decomposition (3)

Granularity: chunk of BS loop iterations

- **Option 1:** requires loop transformation

```
void vector_add(int *A, int *B, int *C, int n) {
   int BS = ...
   for (int ii=0; ii< n; ii+=BS)
       #pragma omp task
       for (int i = ii; i < min(ii+BS, n); i++)
           C[i] = A[i] + B[i];
}
void main() {
   ...
   #pragma omp parallel
   #pragma omp single
   vector_add(a, b, c, N);
   ...
}
```

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

Task Generation Control

Iterative task decomposition (4)

- **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];
}
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 $min(m, n)$

Task Generation Control

Iterative task decomposition: 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.

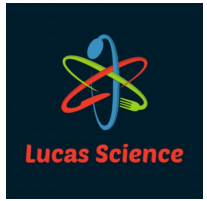
Task Generation Control

`firstprivate` : Specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable, because it exists before the parallel construct.

`private` : Specifies that each thread should have its own instance of a variable.

Instructor Social Media

Youtube: Lucas Science



Instagram: lucaasbazilio

Twitter: lucasebazilio