Exercise Session 1 : Problem Analysis

1 Functional Decomposition

1.1 Task Decomposition

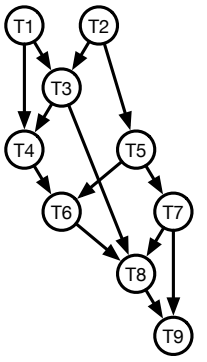
Let us consider the following code:

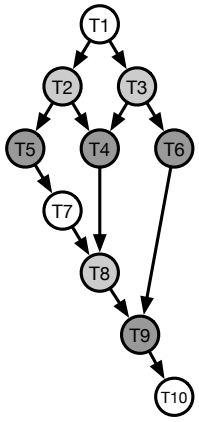
```
a = 0.;
for (size_t i = 0; i < N; i++) {
  a = a + x[i];
}
b = 0.;
for (size_t i = 0; i < N; i++) {
  b = b + y[i];
}
for (size_t i = 0; i < N; i++) {
  z[i] = x[i]/b + y[i]/a;
}
for (size_t i = 0; i < N; i++) {
  y[i] = (a+b) * y[i];
}
```

1. Propose a logical decomposition of the code into tasks without code alteration (e.g., do not break loops).
2. Draw the task dependency graph. For each node, give the computational cost. For each edge, precise the type of the dependency.
3. Compute the critical path length, the maximum degree of concurrency and the average degree of concurrency.
4. Calculate the computational cost (in flop).
5. Analyze what can be run in parallel inside each task.
6. Calculate the speed-up that can be achieved with p processors (we suppose that N is a multiple of p and that p is a multiple of 2).

1.2 Task Graph Analysis

Let us consider the following task dependence graphs all computational costs of the left graph are 1 and all computational costs of the right graph are provided in the table:





T1, T7, T10	$N^3/3$
T2, T3, T8	N^3
T4, T5, T6, T9	$2N^3$

1. For the left graph, identify the critical path, compute the critical path length, the maximum degree of concurrency and the average degree of concurrency.
2. Same questions for the right graph.

2 Data Decomposition

2.1 Data Dependencies

For each code below, determine whether there exist data dependencies according to Bernstein conditions, precise their types and suggest code alterations to remove them if possible.

1.
```
for (i = 1; i < N-1; i++) {
  x[i+1] = x[i] + x[i-1];
}
```

2.
```
for (i = 0; i < N; i++) {
  a[i] = a[i] + y[i];
  x = a[i];
}
```

3.
```
for (i = N-2; i >= 0; i--) {
  x[i] = x[i] + y[i+1];
  y[i] = y[i] + z[i];
}
```

2.2 Loop Parallelization

Let us consider the following code:

```
double function(double A[M][N]) {
  double sum;
  for (size_t i = 0; i < M-1; i++) {
    for (size_t j = 0; j < N; j++) {
      A[i][j] = 2.0 * A[i+1][j];
    }
  }
  sum = 0.0;
  for (size_t i = 0; i < M; i++) {
    for (size_t j = 0; j < N; j++) {
      sum = sum + A[i][j];
    }
  }
  return sum;
}
```

1. Calculate the computational cost (in flop).
2. Analyze what can be run in parallel inside this code and whether it should be transformed to enable parallelism and/or for parallelization to be more efficient.
3. Calculate the speedup that can be achieved with p processors (we suppose that both N and M are multiples of p).
4. Give an upper value of the speedup (when p grows to infinity) when only the forst part cans be performed in parallel and the second part (computing the sum) has to be run sequentially (e.g., because of numerical stability issues).