# Algoritmos paralelos

João Luís Sobral

9-Junho-2021

# Trabalho – AP

- Objetivo
  - Estudar e comparar a escalabilidade de dois algoritmos paralelos (implementações paralelas em OpenMP)

- Parâmetros da avaliação

| | **Peso** |
|---|---|
| Análise (teórica) dos algoritmos | 10% |
| Estudo experimental dos algoritmos e análise dos resultados (justificação para valores obtidos, perfil de execução, comparação com a análise teórica) | 20% |
| Tempo de execução (ver os tempos “expectáveis” na descrição de cada algoritmo) | 60% |
| Relatório | 10% |

# Final Project/Report

1. Select two case studies from:
   - MatMult, Molecular Dynamics, Sort, Stencil2D, ASP
   - Suggestion: select one “good” and one “bad” parallel algorithm
   - Use the PL input data (code from PL lectures “builds” the input)
2. Develop optimised parallel versions
   - Development/test conditions:
     - Shared Memory: Use OpenMP & gcc 5.3.0 (or gcc 7.2.0)
     - 641 machine with 16 or 32 OpenMP threads
     - Collect results with **OMP_PROC_BIND=true**
     - Median of five executions

- Submit a short report (6 pages + code)
- Deadline: 17/06/2021 23:59 BB submission only
- Grade:
  - implementation performance (60%) – see expected performance
  - global quality (40%) – selected algorithms, report, analysis, etc…

# PL lecture 1 - MatMult

- Develop an high performance MM in **C**
  - Shared Memory: Use **OpenMP** (also SIMD)
  - Use a Matrix of **doubles**, **4096x4096**
  - Suggested plan (use gcc 5.3.0+7.2.0):
    1. **Vectorization** (if necessary use **#pragma omp SIMD**)
       - **Compile with -march=native to generate avx code on gcc**
       - **Align data in memory to 32 bytes for better performance**

         **(e.g., on gcc use:** double A[4096][4096] __attribute__ ((aligned (32)));
    2. **Register tiling**
    3. **Cache tiling**
    4. **Parallelism**

  - Expected performance: 1 - 2 seconds
    - Note: no code was provided in the PL lecture (build from scratch)

# PL lecture 2 – Molecular dynamics

- Parallelize the given MD simulation code
  - Shared Memory: Use OpenMP & gcc 5.3.0 (or gcc 7.2.0)
  - Use input of size of 2
    - Verify the number of computed **interactions**: 130 288 673 for size 2
- Suggested plan
  1. Profile the code execution and identify the function responsible by the force computation
  2. Identify the three lines responsible by the 3rd newton law optimization inside the force computation & the update of global state variables
  3. Start with a naive OpenMP particle-based parallelisation taking care of the data races (check the number of computed **interactions).**
  4. Optimise the parallel implementation

     **Clue:** avoid using OpenMP synchronisation as much as possible

Expected performance: 3 - 6 seconds

- Note: use the code provided in the PL lecture as baseline

# PL lecture 3 - sorting

- Parallelize one of given sort codes
  - Shared Memory: Use OpenMP & gcc 5.3.0 (or gcc 7.2.0)
  - Use input of size of 100 000 000 integer keys with 32 bits

Suggested plan

  - Profile the execution of the given sorts
    - (suggestion: perf stat ./sort variant runs size)
  - Start with a naive OpenMP parallelization of the selected sort
  - Optimize the parallel implementation

- Expected performance: 0.8 - 1.4 seconds
  - Note: use the code provided in the PL lecture as baseline

# PL lecture 4

- Parallelize the given All-Pairs Shortest Paths code
  - Shared Memory: Use OpenMP & gcc 5.3.0 (or gcc 7.2.0)
  - 2000x2000 input matrix

Code:

- **Parallel Floyd's Algorithm**
  - the cost of a path can be expressed by the sum of the costs through an intermediate vertex
    - for k, i, j

$$d_{i,j}^{(k)} = \min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)})$$

```
void do_asp() {
    for(int k=0; k<N; k++)
        for(int i=0; i<N; i++)
            if (i != k) {
                for(int j=0; j<N; j++) {
                    int tmp = tab[i][k]+tab[k][j];
                    if (tmp< tab[i][j]) {
                        tab[i][j] = tmp;
                    }
                }
            }
}
```

Clue: the inner loop can be vectorised

Expected performance: 0.13 – 0.45 seconds
Note: use the code provided in the PL lecture as baseline

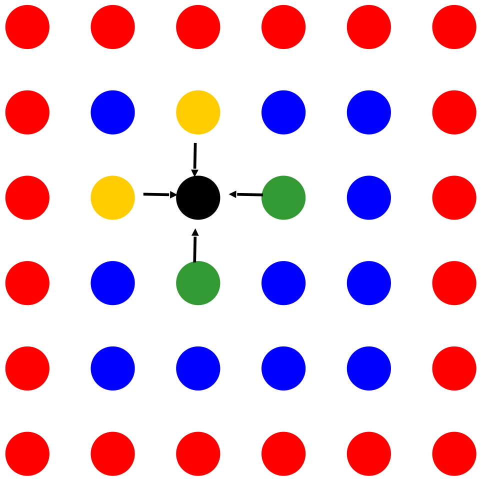
# Algoritmos Paralelos

## Exemplo: algoritmo “stencil2D”

```
for(int i=1; i<N-1; i++) {

    for(int j=1; j<N-1; j++)

        G1[i][j]= 0.2 * (G2[i-1][j]+
                         G2[i+1][j]+
                         G2[i][j-1]+
                         G2[i][j+1]+
                         G2[i][j]);
    }
}
... // copy G1 to G2
```



- 2000x2000 input matrix of type **double**
- Repeat for 1000 iterations (fixed at compile time)

- Expected performance: 2 - 4 seconds
  - **Clues**: avoid matrix copy, enable vectorisation
  - Note: no code was provided in the PL lecture (build from scratch)