Computação Paralela

Optimising program performance on shared memory programming (OpenMP)

João Luís Sobral
Departamento do Informática
Universidade do Minho

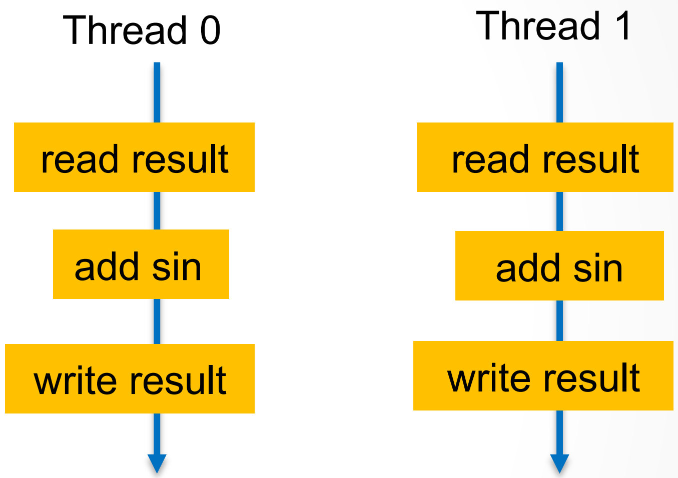
Oct/2024

Recap: Data races (and race conditions)

- A data race **can happen** when two or more threads access (write!) to a shared memory position

```c
int main(){
    double result={0};

    #pragma omp parallel for shared(result)
    for(int i=0; i<1000000;i++) {
        result+=sin(i);
    }
    printf("%f",result);
}
```



- Data races are a sub-set of a broader set of Race Conditions:
  - "a **race condition** is a condition of a program where its behavior depends on relative timing or interleaving"
  - Example:

```
result = f(a);
g(result);
```

Recap: OpenMP Synchronization

- Solving data races with OpenMP **critical** VS **atomic**

```c
int main(){
    double result={0};

    #pragma omp parallel for shared(result)
    for(int i=0; i<1000000;i++) {
        result+=sin(i);
    }
    printf("%f",result);
}
```

Data race

.L4: ..
call sin
vmovsd ~~8(%rsp~~%r12), %xmm1 ; load result into %xmm1
vaddsd %xmm0, %xmm1, %xmm1 ; add
vmovsd %xmm1, ~~8(%rsp~~%r12) ; update result
...
jne .L4

- **Critical**: coarse-grain and pessimist

...
call GOMP_critical_start
vmovsd ~~8(%rsp~~%r12), %xmm1 ; load result into %xmm1
vaddsd %xmm0, %xmm1, %xmm1
vmovsd %xmm1, ~~8(%rsp~~%r12) ; update results
call GOMP_critical_end

Mutual exclusion: ensures that only one thread executes the critical region (e.g., blocks the calling thread if another thread is still executing the region)

- **Atomic**: fine-grain and optimistic (note: simplified code)

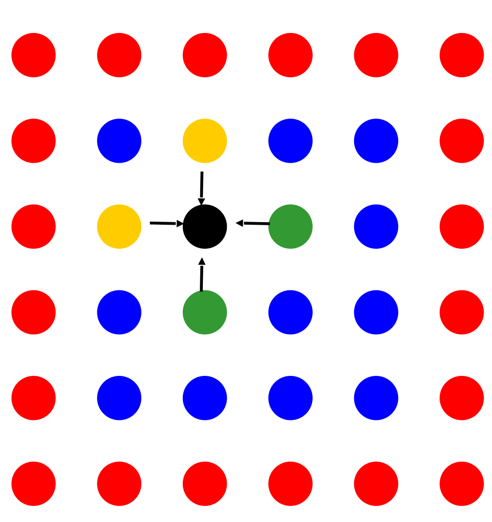
.L4 ..
call sin
.L5: vmovsd (%r12), %xmm1
vaddsd %xmm0, %xmm1, %xmm1
...
lock cmpxchgq %xmm1, (%r12) ; atomic comp & exchange
cmpq %rax, %rdx
jne .L5 ; repeat if update failed
...
jmp .L4

Repeat until successful update

Race conditions and data dependencies

Example: stencil computation

```
for(i=1, i<N-1, i++ )
  for(j=1, j<N-1, j++ )
        A[i,j] = 0,2 x (      A[i-1,j] +
                    A[i,j-1]  + A[i  ,j] + A[i,j+1]
                              + A[i+1,j]  );
```



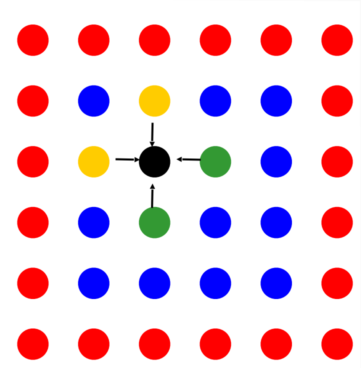
- **Instruction level Parallelism (ILP)**
  - Read multiple values of A[..,..] from memory in parallel?
  - Perform multiple arithmetic operations in parallel (which?)
  - Multiply (by 0,2?) and write A[i,j] after all operations are done
  - **How to improve ILP?**
    - Can we compute $A_{i,j}$ and $A_{i,j+1}$ in parallel?
  - **What dependency constrains the computing of multiple elements of A in parallel?**

```
A[i,j+1] = 0,2 x (      A[i-1,j+1] +
A[i,j]   + A[i  ,j+1] + A[i,j+2]
         + A[i+1,j+1]  );
```

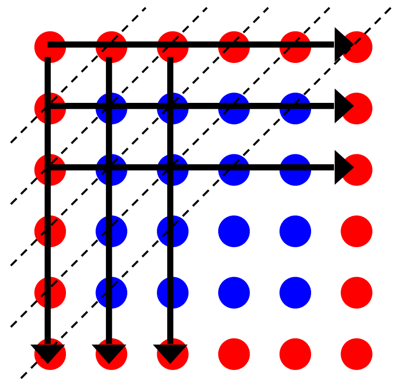
Race conditions and data dependencies

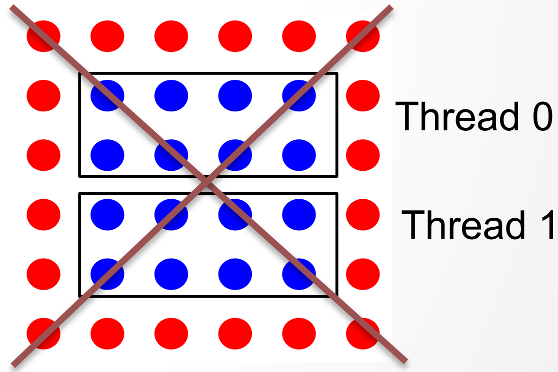
Example: stencil computation

```
for(i=1, i<N-1, i++ )
  for(j=1, j<N-1, j++ )
        A[i,j] = 0,2 x (      A[i-1,j] +
                 A[i,j-1]  + A[i  ,j] + A[i,j+1]
                           + A[i+1,j]  );
```



- **Thread-level [task] parallelism**
  - Each thread [task] computes a set of rows of the matrix
  - Data dependencies?





Does not respect data dependencies (introduces a **race condition**)

WA – Phase 2

1. Increase the value of N

2. Optimised solver

   a) red & **black** phases

   b) detects early convergence

```
void lin_solveRedBlack(int M, int N, int O, int b, float *x, float *x0, float a, float c) {
    float tol = 1e-7, max_c, old_x, change;
    int l = 0;

    do {
        max_c = 0.0f;
        for (int i = 1; i <= M; i++) {
            for (int j = 1; j <= N; j++) {
                 for (int k = 1 + (i+j)%2; k <= O; k+=2) {
                    old_x = x[IX(i, j, k)];
                    x[IX(i, j, k)] = (x0[IX(i, j, k)] +
                                      a * (x[IX(i - 1, j, k)] + x[IX(i + 1, j, k)] +
                                           x[IX(i, j - 1, k)] + x[IX(i, j + 1, k)] +
                                           x[IX(i, j, k - 1)] + x[IX(i, j, k + 1)])) / c;
                    change = fabs(x[IX(i, j, k)] - old_x);
                    if(change > max_c) max_c = change;
                }
            }
        }
        for (int i = 1; i <= M; i++) {
            for (int j = 1; j <= N; j++) {
                for (int k = 1 + (i+j+1)%2; k <= O; k+=2) {
                    old_x = x[IX(i, j, k)];
                    x[IX(i, j, k)] = (x0[IX(i, j, k)] +
                                      a * (x[IX(i - 1, j, k)] + x[IX(i + 1, j, k)] +
                                           x[IX(i, j - 1, k)] + x[IX(i, j + 1, k)] +
                                           x[IX(i, j, k - 1)] + x[IX(i, j, k + 1)])) / c;
                    change = fabs(x[IX(i, j, k)] - old_x);
                    if(change > max_c) max_c = change;
                }
            }
        }
        set_bnd(M, N, O, b, x);
    } while (max_c > tol && ++l<20);
}
```

Performance of parallel applications

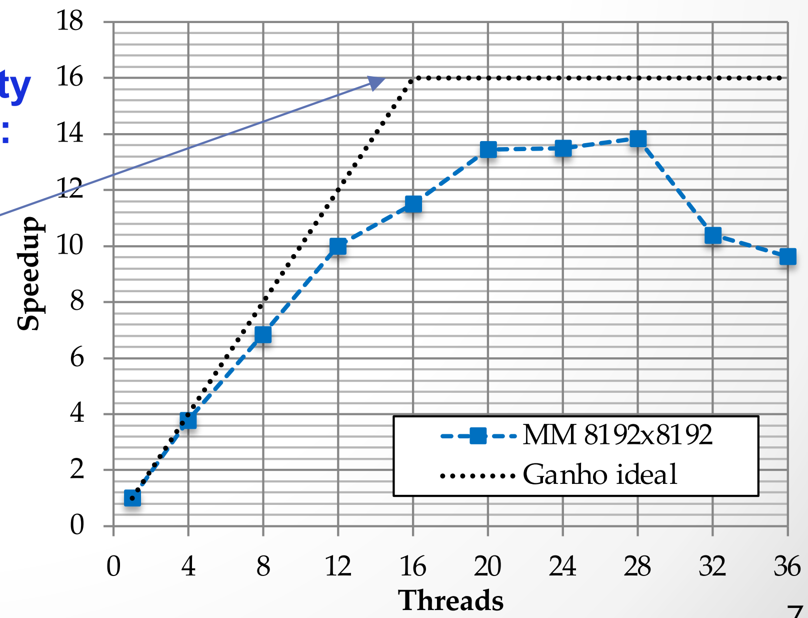
What is the definition of performance?

- Multiple alternatives:
  - Execution time, efficiency, scalability, memory requirement, throughput, latency, project / development costs, portability, reuse potential
  - The relevance of each one depends on the concrete case
  - The most common measure in parallel applications is **execution time**

- **Scalability analysis** of parallel applications:
  - **speedup (gain):** execution time of the **best** sequential implementation / execution time of the parallel version
  - **Strong scalability** analysis:
    - Speedup increase with PU for a fixed problem data size
    - ideal speedup is proportional to the number of assigned <u>physical</u> PUs

    (system with 16 physical PUs in this example)
  - **Weak scalability** analysis:
    - Increase problem data size as the number of PU increases
    - Ideally the execution time should remain constant

Strong scalability example:



Performance of parallel applications

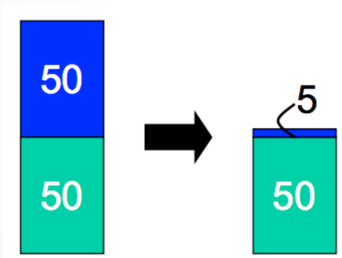
Amdahl's law (and its impact on the strong scalability)

- The sequential execution time can be divided into:
  - Time doing non-parallelizable work (serial work)
  - Time doing parallelizable work
- The fraction of non-parallelizable work (serial fraction of work) limits the maximum speedup
  - *P* – number of PU (e.g., #cores)
  - *f* – serial fraction of work
  - S_p – speedup

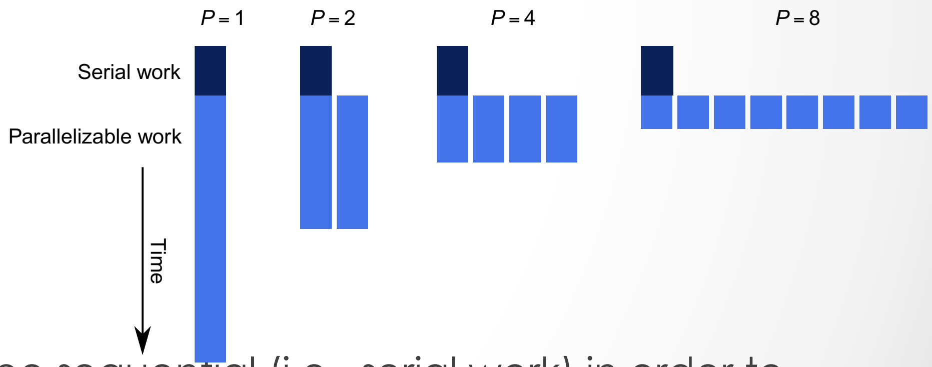
$$S_P \le \frac{1}{f+(1-f)/P}$$

The maximum speedup is:
1 / serial fraction of work

- Example (f=0.5):



10x speedup in parallelizable work results in 1.8x overall speedup



- What fraction of the original computation can be sequential (i.e., serial work) in order to achieve a speedup of 80 with 100 PUs?
  - 80 = 1/ (f+(1-f)/100) ⇔ f = 0.0025 (e.g., 0.25%)
- Reinforces the idea that we should prefer algorithms that are suitable for parallel execution: ***think parallel!***

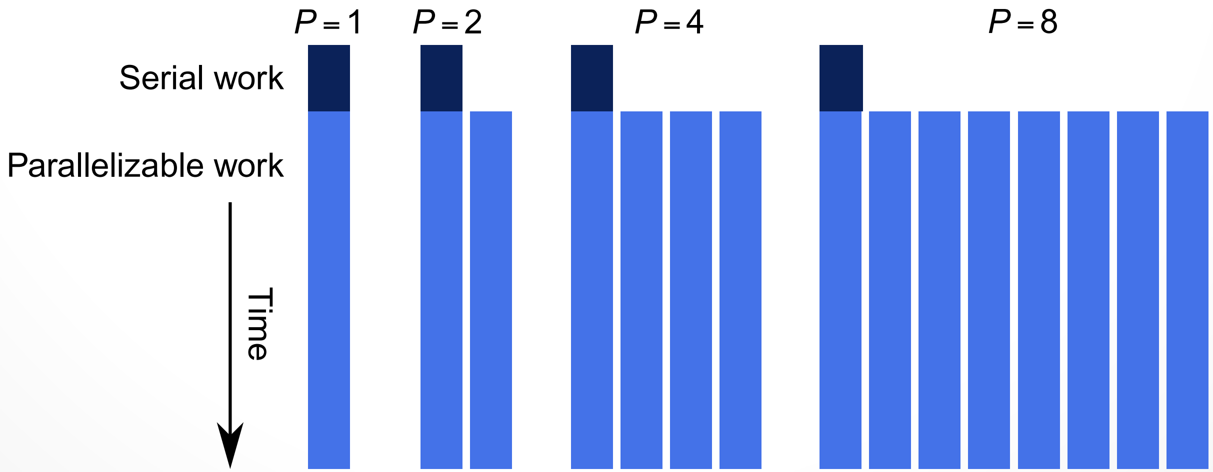
Performance of parallel applications

Speedup anomalies

- Super-linear speedup (superior to the number of PUs):
  in most cases it is due to cache effects

Gustafson's law (aka weak scalability analysis)

- Increase problem size as the number of PU increases
  - Larger computational resources are usually devoted to larger problem sizes
- The fraction of serial work generally decreases with the problem size
- Weak-scaling example (with ideal speedup)



Performance of parallel applications

Experimental study

- **Sequential execution profile**:
  - Identify application **hot-spots**
    - Functions that take most of the time to execute
  - Can be implemented by specific tools or by directly instrumenting the code
    - There is always an overhead introduced in the base application
- **Parallel execution profile**:
  - Gathers per-thread performance data
  - More difficult to interpret
- **Hot-spots** can change as the application is improved
  - e.g., by introducing parallelism into an hot-spot, other place can become hot-spot

Generic causes of lack of scalability
(in shared memory programming)

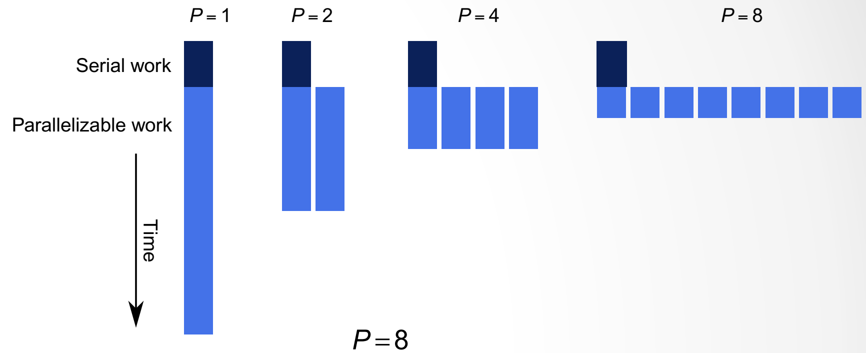
Why parallel applications do not have an ideal speedup (1)?

1. Serial work (Amdahl's law)

Computations are preformed serial

non parallelizable work

serialized calls to functions (e.g., rand)

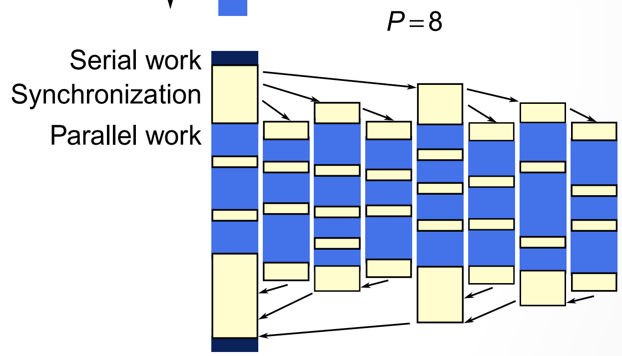


1. Parallelism overhead

Additional operations on parallelizable work

thread/task management,

redundant computations, ...

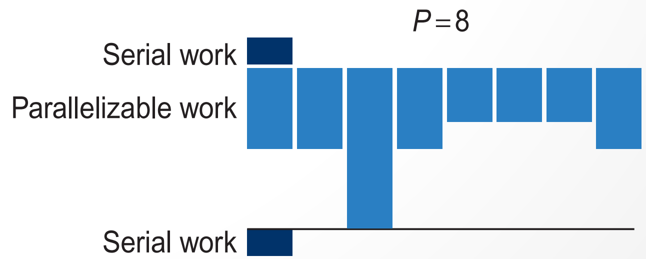


2. Idle time

Some PUs remain idle while others are still performing computations

load imbalance,

waiting on a synchronisation point, …



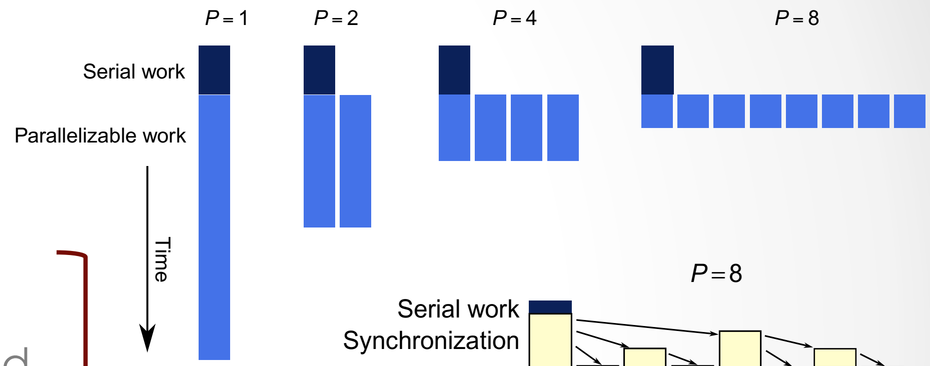
NOTE: execution time is defined by the slowest PU

Scalability problems in shared memory

Why parallel applications do not have an ideal speedup (2)?

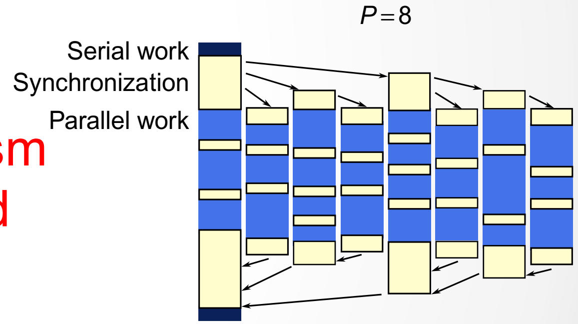
1. **% of serial work (Amdahl's law)**
2. **Memory wall**
   - Serializes memory accesses

Serial work



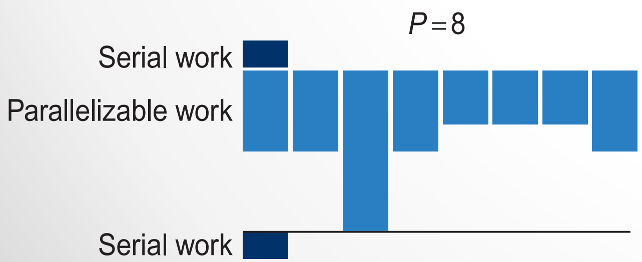
3. **Parallelism/task granularity**
   - Small tasks introduce more (relative) overhead
4. **Synchronisation overhead**
   - Might also serialize execution (e.g. critical)
   - Includes (serial) calls to external routines (e.g., malloc)

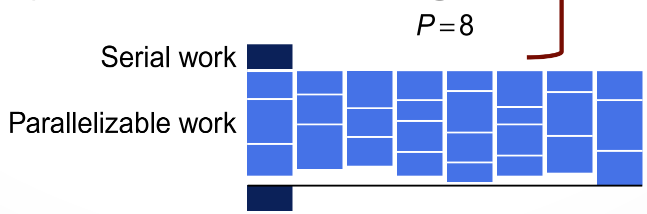
Parallelism overhead



5. **Load imbalance**
   - Over-decomposition can improve load balancing, **but ...**

Serial work / idle time



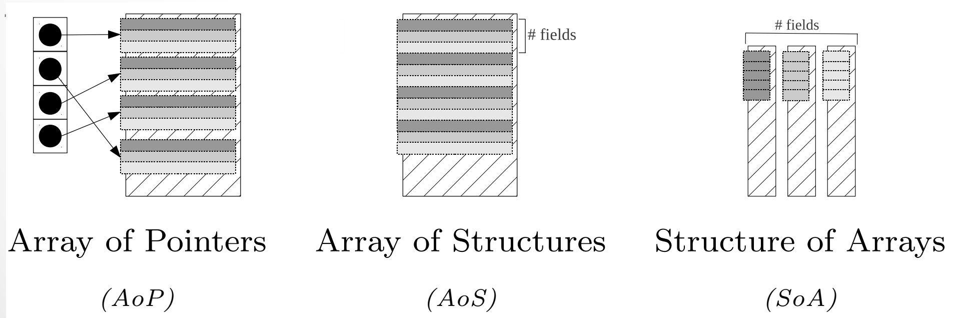


Scalability problems in shared memory

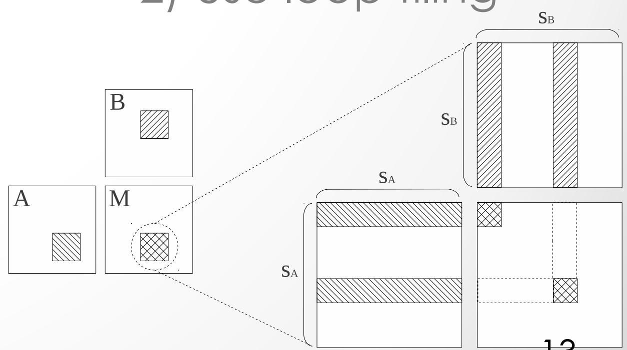
Some reasons for the lack of scalability *(1)*

2. **Memory wall:** how to identify memory or cache bandwidth limitation
   - Diagnostic (some options):
     - Theoretical analysis: roofline model extended to multicore systems
     - (simpler) based on experimental measures:
       1. required memory bandwidth (per core) vs available bandwidth
       2. simple estimation of the arithmetic intensity: #I / LLC.MISS (or L2.MISS)
       3. CPI increase with the number of threads (increase in cycles waiting for memory)
   - Action:
     - Improve data locality
   - Approaches

1) Data layout: Convert AOP to AOS/SOA layout



2) Use loop tiling



Scalability problems in shared memory

Some reasons for the lack of scalability *(2)*

3. Parallelism/task granularity: fine-grained parallelism

- Diagnostic:
  - Measure task granularity (computation/parallelism ratio)
    (#I seq *vs.* sum #I par)
- Action:
  - Increase task size or management overhead to reduce parallelism overhead
- Approaches:
  - Favour static loop scheduling (in certain cases must be explicitly implemented)
  - Decrease task creation frequency

old OpenMP versions

```
# pragma omp parallel for
for(int i = 0;  i<100; i++)
        …

#pragma omp parallel for
for(int j= 0; j<100; j++)
        …
```

→

```
# pragma omp parallel {
        …
    #pragma omp for
    for(int i = 0;  i<100; i++)
        …
    #pragma omp for
    for(int j= 0; j<100; j++)
        …
}
```

Scalability problems in shared memory

Some reasons for the lack of scalability *(3)*

4. **Excessive task synchronisation** (due to dependencies)
   - Diagnostic:
     - (?) Run task without synchronisation (producing wrong results!)
   - Action
     - Remove synchronisation or use more efficient alternatives
   - Approaches
     - Increase task size
     - Speculative/redundant computations
     - Use thread local values (caution with false sharing of cache lines)

```
sum = 0;
# pragma omp parallel for
for(int i = 0;  i<100; i++) {
# pragma omp atomic
       sum += array[i];
}
```

→

```
sum = 0;
# pragma omp parallel for reduction(+:sum)
for(int i = 0;  i<100; i++) {
          sum += array[i];
}
```

Scalability problems in shared memory

Some reasons for the lack of scalability *(4)*

4. Load imbalance (due to dependencies)

- Diagnostic:
  - Measure each task computational time (#I / per thread)
- Action
  - Improve scheduling/mapping
- Approaches
  - Cyclic/dynamic/guided scheduling
  - Custom (static) loop scheduling

```
# pragma omp parallel for
for(int i = 0;  i<100; i++) {
        …
}
```

→

```
# pragma omp parallel {

    int myid = omp_get_thread_num();
    int nthreads = omp_get_num_threads()

    // cyclic scheduling
    for(int i = myid;  i<100; i+=nthreads) {
        …
    }
}
```

Scalability problems in shared memory

Summary:

Possible metrics to present

1. **% of serial work**

2. **Memory bandwidth and arithmetic intensity**
   - data locality optimisations

3. **Task granularity / parallelism overhead**
   - increase granularity

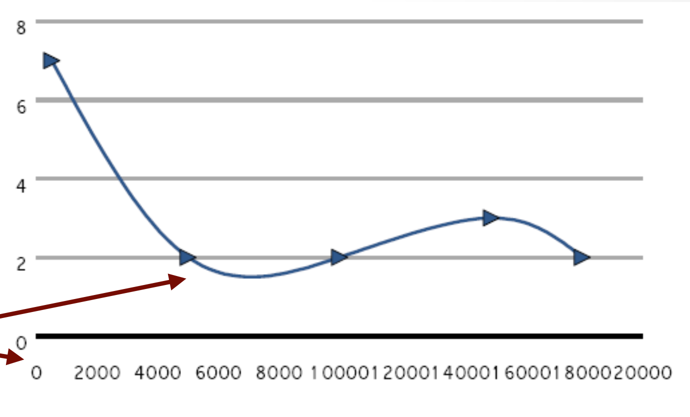
4. **Synchronisation overhead**
   - measure programs without synchronisation / decrease dependencies

5. **Compute time per parallel task**

Measuring performance

- **Presenting results**
  - ➢ Present results in a <u>readable</u> (& compact) format

Tempos de Execução				
	Nº de Clientes no Ficheiro			
Operações	5000	10000	15000	18000
Carregar Dados	10.019 ms	20.881 ms	32.027 ms	40.992 ms
Inserir Cliente	7.100 μs	7.400 μs	8.800 μs	9.500 μs
Procura por Nome	0.360 μs	0.380 μs	0.400 μs	0.430 μs
Procura por Nif	0.020 μs	0.020 μs	0.020 μs	0.020 μs
Percorrer Estrutura	0.092 ms	0.232 ms	0.470 ms	0.673 ms



  - ➢ Place clear legends in tables and plots
  - ➢ Do not extrapolate values
    - Use the right number of significant digits: **1,00004 s !**
  - ➢ Use constant increments in X axis and Y axis
    - Scales can lead to wrong conclusions!
      - Use lin-lin or log-log on both axis (prefer X-Y plots)
    - Represent 0 (or 1)
  - ➢ Justify obtained results
    - Investigate/comment unexpected values

Measuring performance

- **Some common errors**

  - **Not documenting experimental environment / including irrelevant details**

    Temperatura do processador: Esteve sempre contida no intervalo [48ºC,54ºC],

  - **Not repeating the experience**
    - Reduces the impact of the OS, garbage colector, etc..

  - **Time spent to serve interruptions & for debugging**
    - Disk reads (due to page faults, …)
    - "`printf`"

  - **Not considering timer reading <u>overhead</u> / <u>resolution</u>**
    - Insertion takes 0 ???
      - **Solution:** Measure multiple operations

Procurar NIF	1	2	1	1	1	1	2	2	1	1

1 microsecond is
the clock resolution

  - **Cold/warm cache** (and JIT in Java)