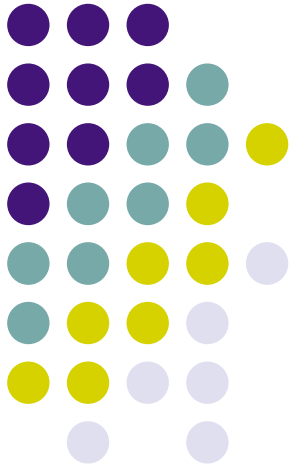
Paradigmas de Computação Paralela

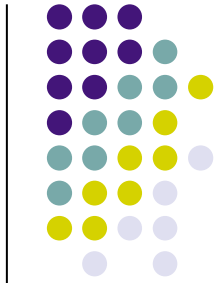
Measuring and optimising performance on shared memory (OpenMP)

João Luís Sobral
Departamento do Informática
Universidade do Minho

24 October 2017



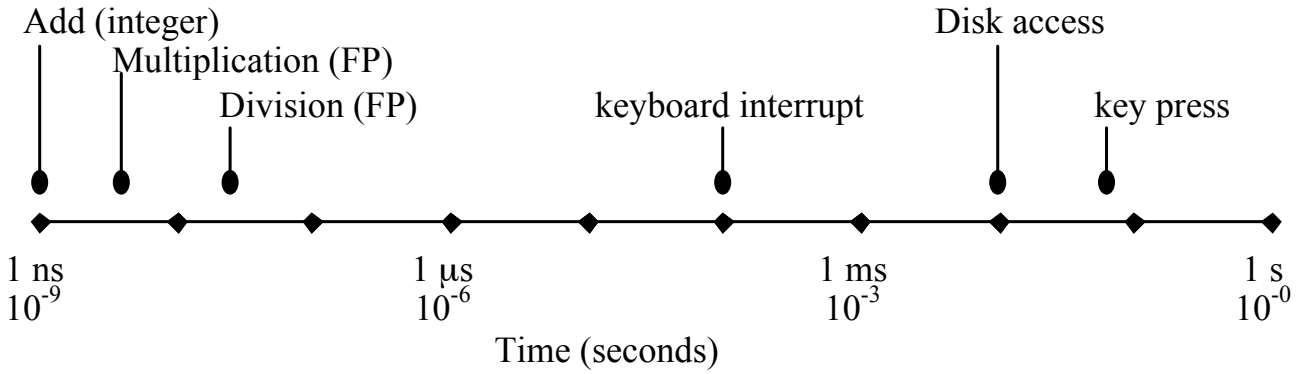
Measuring performance



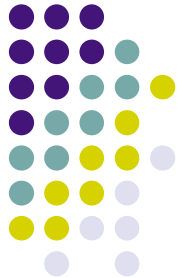
Principles

- **Isolate from external factors**
  - **Consider the measurement overhead**
  - **Repeat the measurement**
- **Document the experiment to be reproducible by others**
  - **Hardware, software versions, system state…**
- **Important: clock resolution**
  - **Precision: difference between measured and real time**
  - **Resolution: time unit between clock increments**
    - **In principle, it is not possible to measure events shorter than the clock resolution, but…**

Event timescale (1GHz machine)



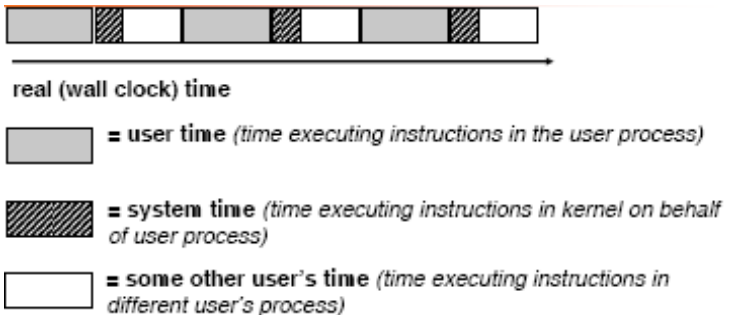
Measuring performance

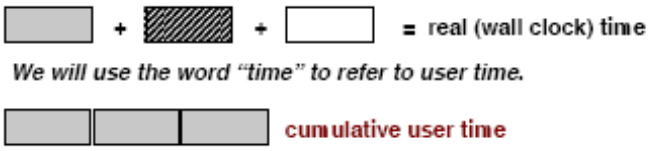


How much time is required to execute an application?

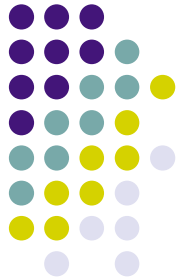
- **CPU time**
  - Time dedicated exclusively to program execution
  - Does not depend on other activities in the system
- **Wall time**
  - Time measured since the start until de end of the execution
  - Depends on the system load, I/O, etc.

- **Complexities**
  - Process scheduling (10ms?)
  - Load introduced by other processes (e.g., garbage collector in JVM, other users)





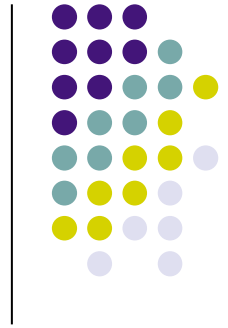
Measuring performance



How to combine results from several measurements?

- **Average**
  - **Affected by extreme high/low values**
  - **Also include the deviation among measurements**
- **Best measure**
  - **Value in ideal conditions**
- **Average of k-best**
  - **Removes outsiders**
- **Median**
  - **More robust to large variations**

Measuring performance



Options for time measurement

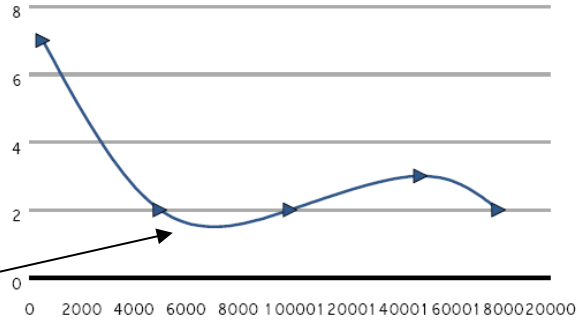
- "Time" command line
  - Only for measurements >>1seg
- gettimeofday()
  - Returns the number of microseconds since 1-Jan-1970
  - Uses the "Timer" or the cycle counter (depends on the platform)
  - Best case resolution: 1us
- Cycle counter
  - High resolution
  - Useful for measurements <<1s
- Timer function in OpenMP / MPI
  - *omp_get_wtime, omp_get_wtick*
  - *MPI_Wtime*
- System.nanoTime() in Java 4

Measuring performance

Presenting results

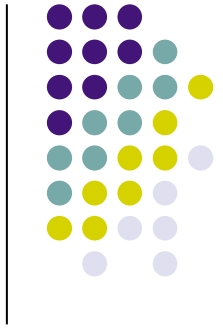
- **Present results in a readable (compact) manner**

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 Legends in tables and graphs**
- **Do not extrapolate values**
  - Number of significant digits: **1,00004 s!**
- **Attention to axis scales (can lead to wrong conclusions!)**
  - Do not use constant increments (in X axis)
    - Use lin-lin or log-log on both axis (X-Y graphs)
  - Do not represent 0
- **Justify obtained results**
  - Investigate/comment unexpected values

Measuring performance



Common errors

- **Do not document experimental environment / include irrelevant details**

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

- **Do not repeat the experience**
  - **Reduces the impact of the OS, garbage colector, etc..**

- **Include I/O time**
  - **Disk reads**
  - **“printf”**

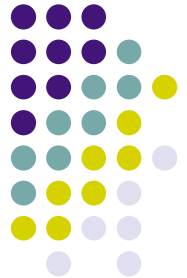
- **Do not consider timer reading overhead / resolution**
  - **Insertion takes 0???**
    - **solution: Measure multiple operations**

- **Do not warm the cache (and JIT in Java)**

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

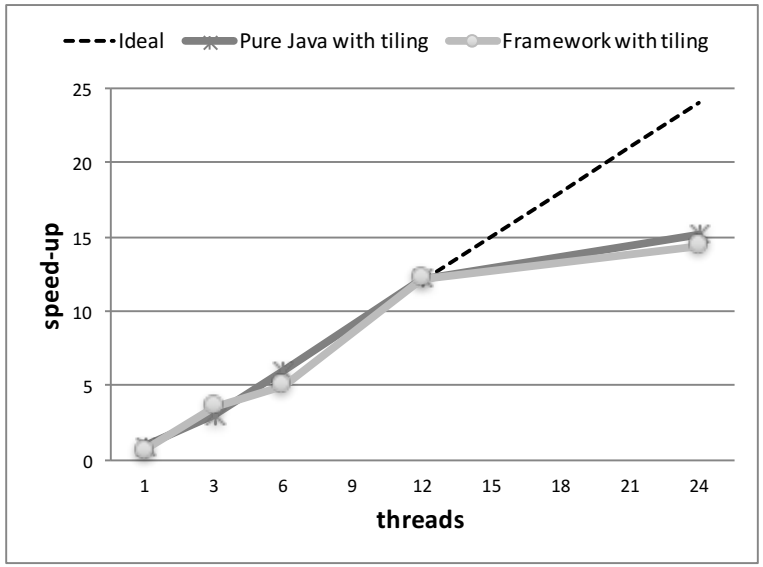
1 microsecond is the clock resolution

Performance of parallel applications

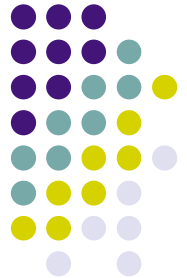


What is the definition of performance?

- There are multiple alternatives:
  - Execution time, efficiency, scalability, memory requirement, throughput, latency, project costs / development costs, portability, reuse potential
  - The importance of each one depends on the concrete application
- Most common measure in parallel applications (*speed-up*)
  - tseq/tpar



Performance of parallel applications



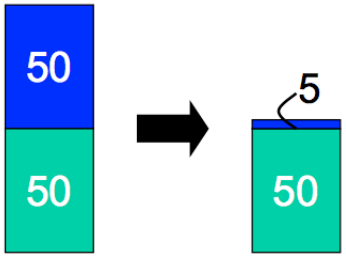
Amdahl's law

- The sequential component of an application limits the maximum speed-up
  - *S* – number of resources (e.g., cores)
  - *P* – faction that runs in parallel

$$\frac{1}{(1-p)+\frac{p}{s}}$$

Maximum speed-up = 1/(1-p)
(1/sequential fraction)

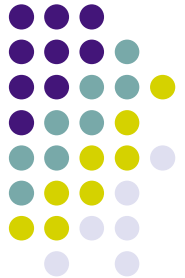
- Example:



10x increase results in
1,8x overall increase

- What fraction of the original computation can be sequential in order to achieve a speedup of 80 with 100 processor?
  - 80 = 1/ (1-p +p/100) ⇔ p = 0,9975
- Reinforces the idea that we should prefer algorithms suitable for parallel execution: *think parallel*.

Performance of parallel applications



Experimental study and evaluation of implementations

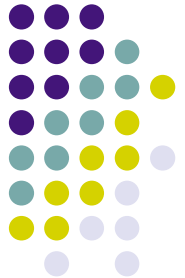
- Parallel computing has a strong experimental component
  - Many problems are too complex for a realization only based on models
  - Performance model can be calibrated with experimental data

- How to ensure that result are precise and reproducible?
  - Perform multiple experiments and verify clock resolution
  - Results should not change among in small difference: less than 2-3%

- Execution profile:
  - Gather several performance data: number of messages, data volume transmitted
  - Can be implemented by specific tools or by directly instrumenting the code
    - There is always an overhead introduced in the base application

- *Speed-up anomalies*
  - superlinear (superior to the number of processors) – in most cases it is due to cache effects

Performance of parallel applications



Techniques to measure the application time-profile (*profiling*)

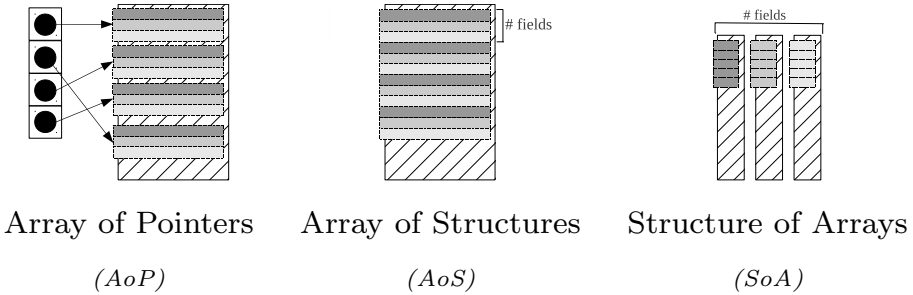
- **Polling**
  - the application is periodically interrupted to collect performance data
  - Example: gprof
- **Instrumentation**
  - code is introduced (by the programmer or by tools) to collect performance data about useful events
  - Tends to produce better results but also produces more interference (e.g., overhead)
  - Example: Valgrind (callgrind)

Scalability problems in shared memory

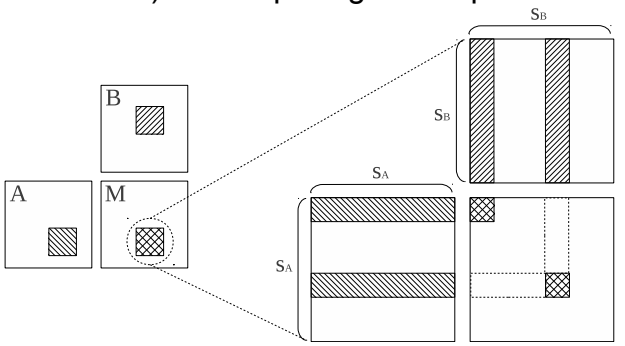
Some reasons for the lack of scalability (1)

- % of computations not performed in parallel (Amdahl law)
- **Memory bandwidth limitation**
  - Diagnostic:
    - Measure required memory bandwidth (per core) and compare against available bandwidth (LLC.MISS * 64 / execution_time)
    - Computational intensity = #I / LLC.MISS
  - Action:
    - Improve locality
  - Approaches

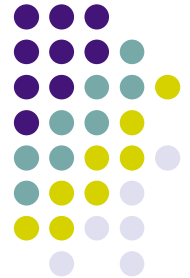
1) convert AOP to AOS/SOA



2) use loop tiling techniques

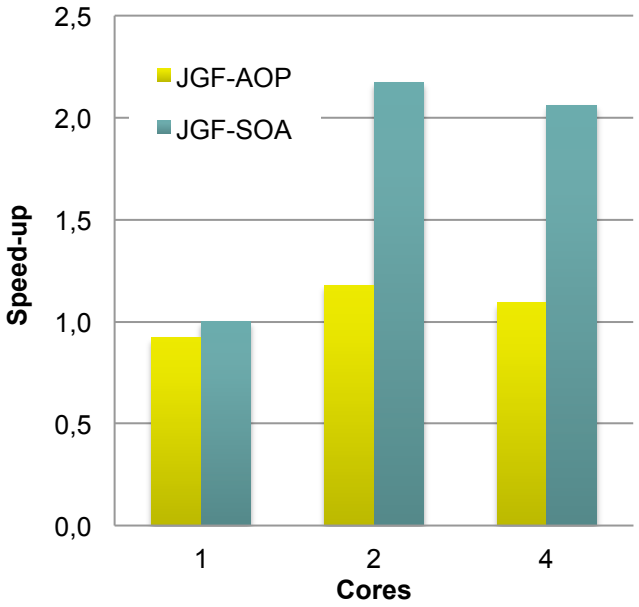


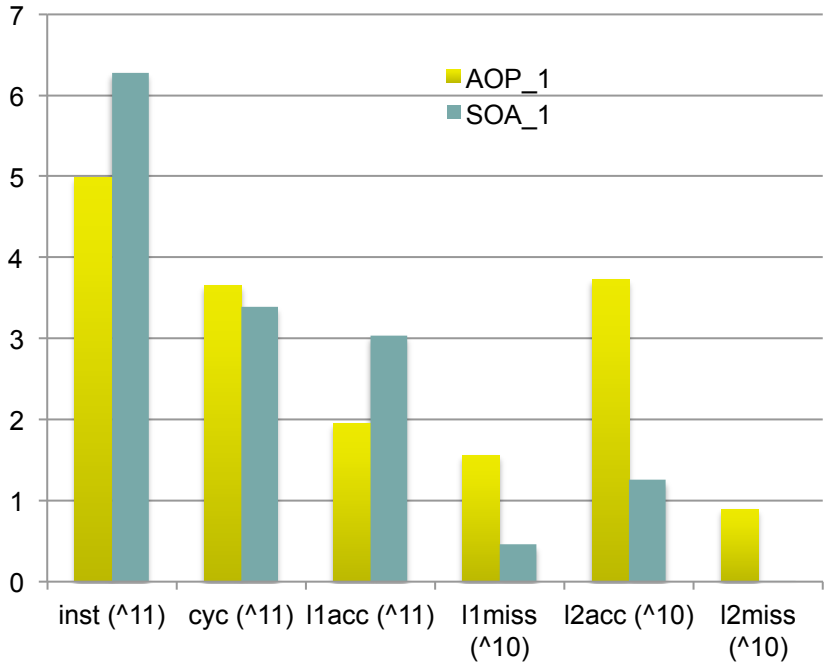
Scalability problems in shared memory



Some reasons for the lack of scalability (1)

- Example: memory bandwidth limitation
  - JGF MD (AOP vs SOA) on a 4-core machine

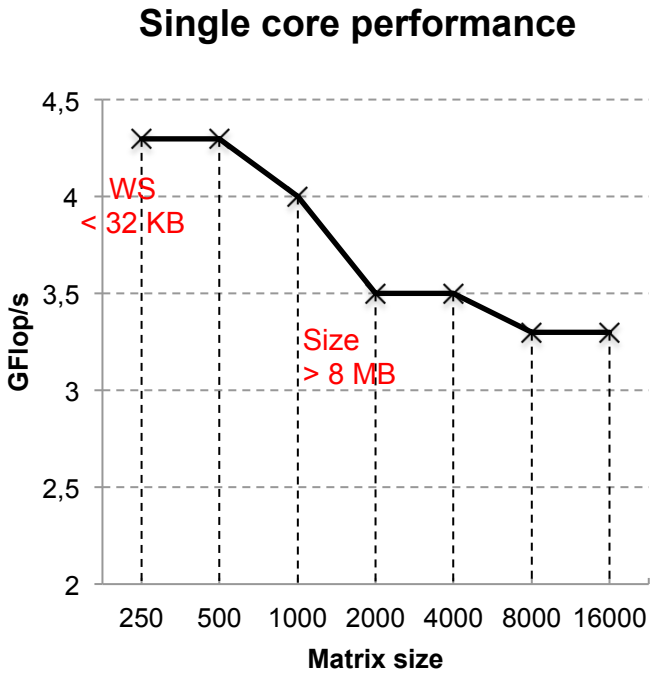


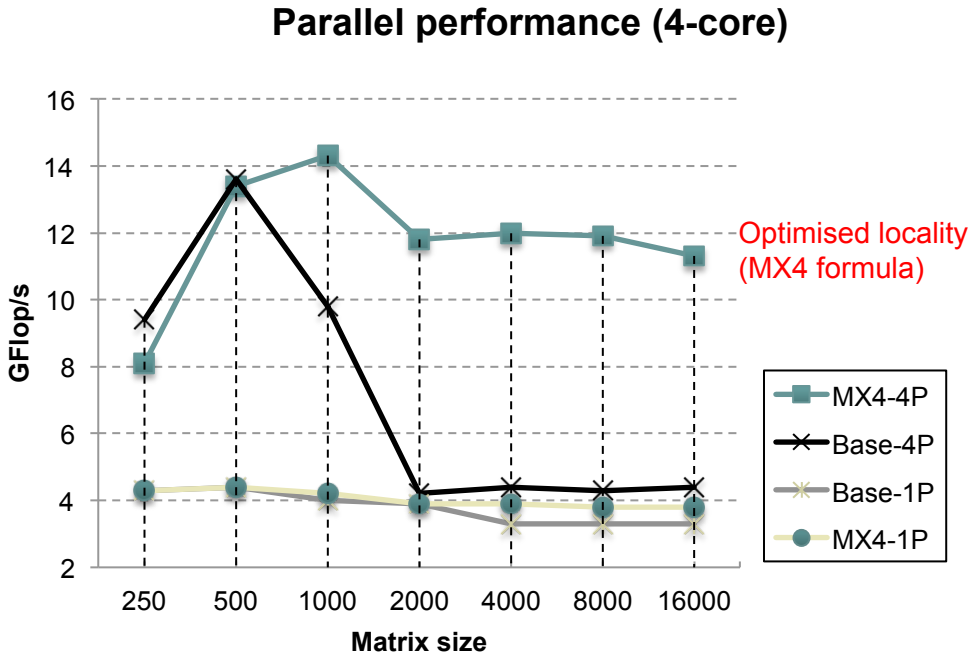


Scalability problems in shared memory

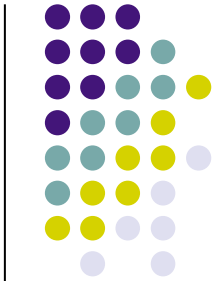
Some reasons for the lack of scalability (1)

- Example: memory bandwidth limitation (cont)
  - SOR Red-black locality (temporal)





Scalability problems in shared memory



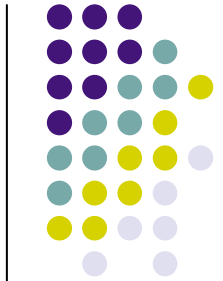
Some reasons for the lack of scalability (2)

- **Fine-grained parallelism** (excessive parallelism overhead)
  - Diagnostic:
    - Compute task granularity (computation/parallelism ratio)
      (#I seq vs sum #I par)
  - Action:
    - Increase task granularity and/to reduce parallelism overhead
  - Approaches:
    - Favour static loop scheduling (in certain cases implemented explicitly)
    - Decrease task creation frequency

```
# 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



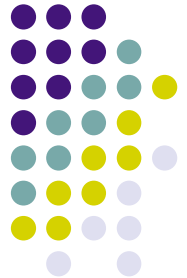
Some reasons for the lack of scalability (3)

- **Excessive task synchronisation** (due to dependencies)
  - Diagnostic:
    - (?) Run task without synchronisation (producing wrong results!)
  - Action
    - Remove synchronisation
  - Approaches
    - Increase task granularity
    - Speculative/redundant computations
    - Use thread local values (caution with false sharing of cache lines / memory usage)

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

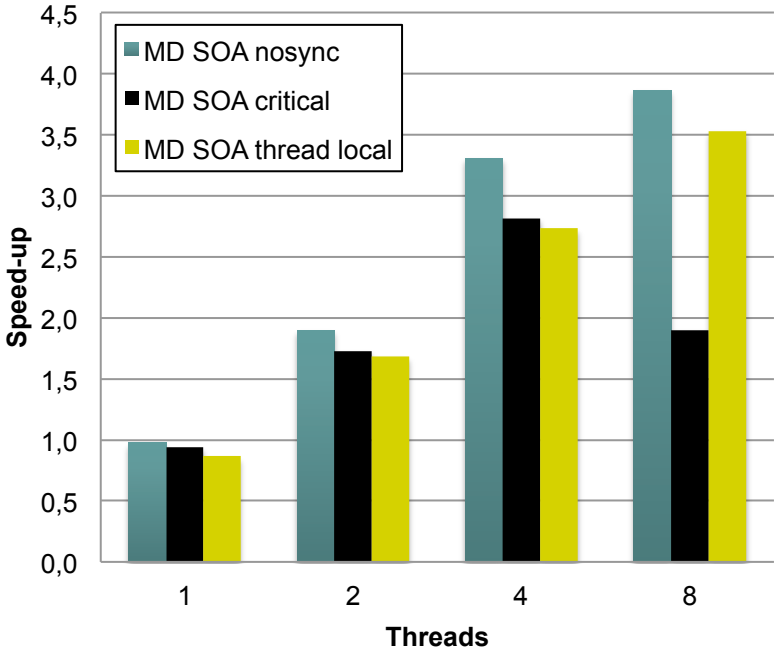
```
sum = 0;
# pragma omp parallel for reduction(+:sum)
for(int i = 0;  i<100; i++) {
                    sum += array[i];
}
```

Scalability problems in shared memory

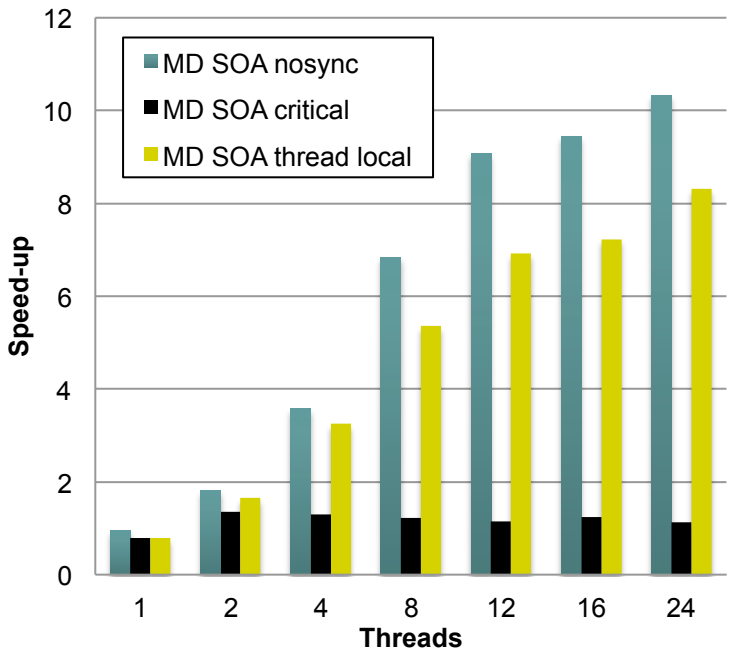


Some reasons for the lack of scalability (3)

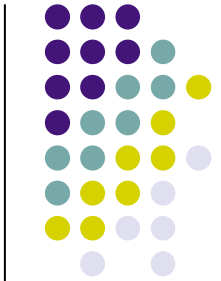
- Example: excessive task synchronisation
  - **Scalability of JGF MD**
  - **4-core i7**



2x6-core i7 NUMA



Scalability problems in shared memory



Some reasons for the lack of scalability (4)

- **Bad load distribution**
  - Diagnostic:
    - Measure each task computational time
  - Action
    - Improve scheduling/mapping
  - Approaches
    - Cyclic/dynamic/guided scheduling
    - Custom (static) loop scheduling

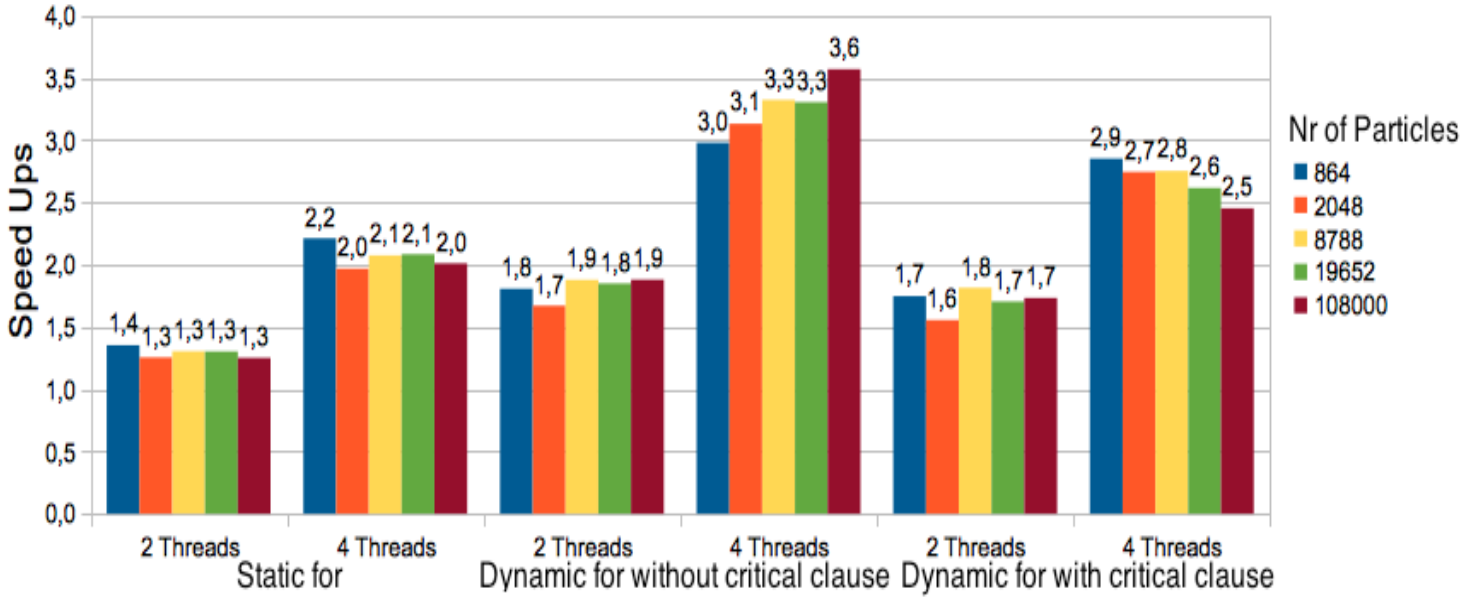
```
# pragma omp parallel for
for(int i = 0;  i<100; i++) {
                …
}
```

```
# pragma omp parallel {

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

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

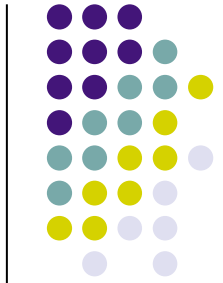
Scalability problems in shared memory

Some reasons for the lack of scalability (4)

- Example: MD load distribution



Scalability problems in shared memory



Possible metrics to present

1. % of parallel code

2. Memory bandwidth and computational intensity
   - locality optimisations

3. Task granularity
   - increase granularity

4. Synchronisation overhead
   - Measure programs without synchronisation / decrease dependencies

5. Measure compute time per parallel task