Lic. Physics Eng.

2021/22

A.J.Proença

Optimizing Performance with OpenMP

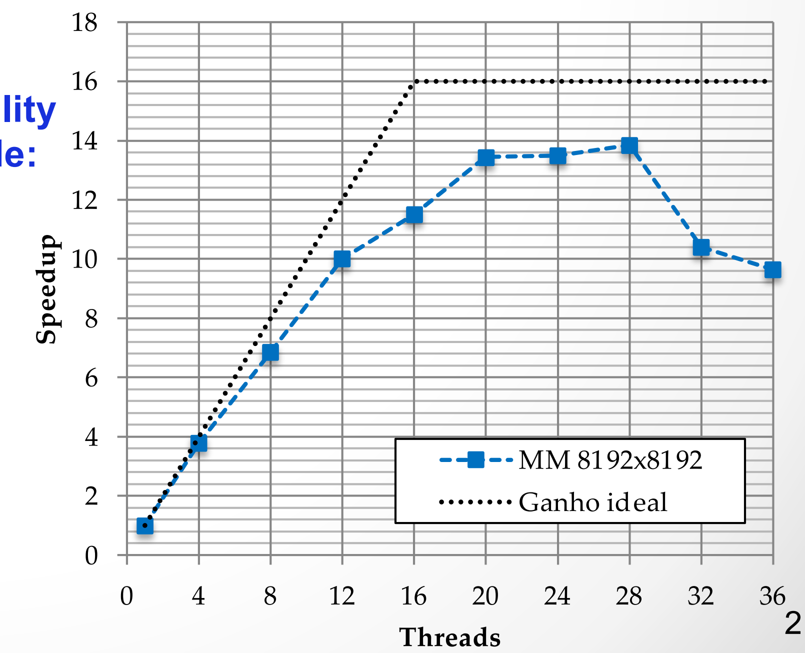
(most slides are from previous year)

Performance of parallel applications

What is the definition of performance?

- Multiple alternatives:
  - Execution time, efficiency, scalability, memory requirement, throughput, latency, project costs / development costs, portability, reuse potential
  - The relevance of each one depends on the concrete application
- Most common measure in parallel applications: execution time or speedup
  - Time of the **best** sequential implementation / time of the parallel version
  - **Strong scalability** analysis:
    - Speedup increase with PU for a fixed problem data size
    - Ideal speedup is proportionally to PU
  - **Weak scalability** analysis:
    - Increase problem data size as the number of PU increases
    - Ideally the execution time should remain constant

Strong scalability example:



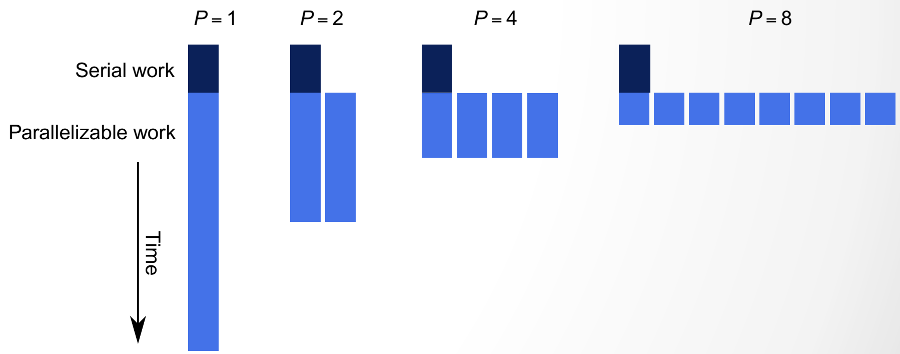
Performance of parallel applications

Amdahl's law (strong scalability analysis)

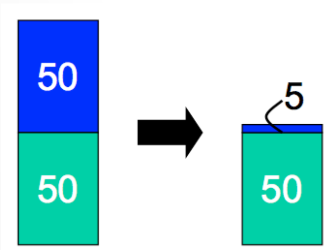
- Measures the execution time of the parallel version (*Tpar*) as the number of PU increases
- *Tseq* can be divided into:
  - Time doing non-parallelizable work (serial work)
  - Time doing parallelizable work
- The fraction of non-parallelizable work (serial work) limits the maximum speedup
  - *P* – number of PU (e.g., #cores)
  - *f* – fraction that runs in serial

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

Maximum speedup = 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 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!***

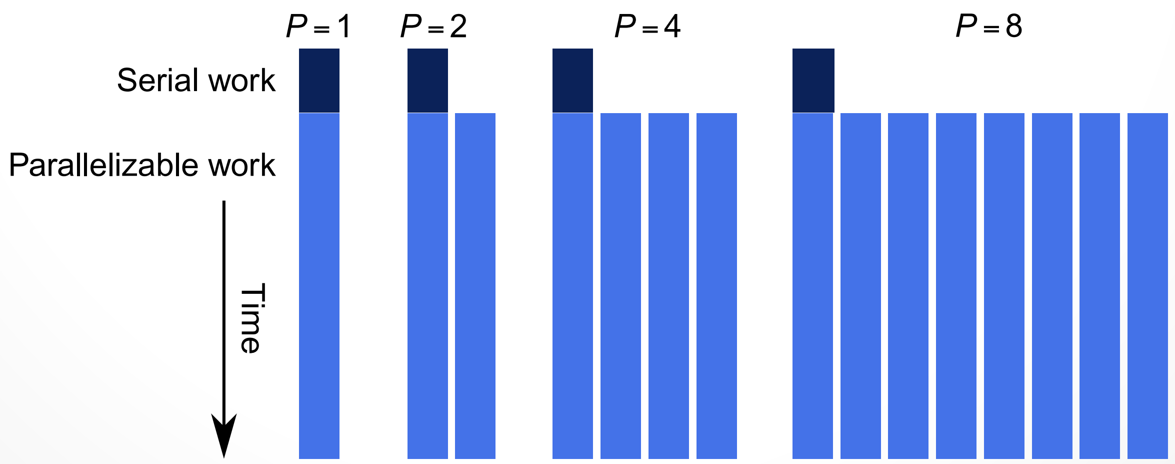
Performance of parallel applications

Speedup anomalies

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

Gustafson's law (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)



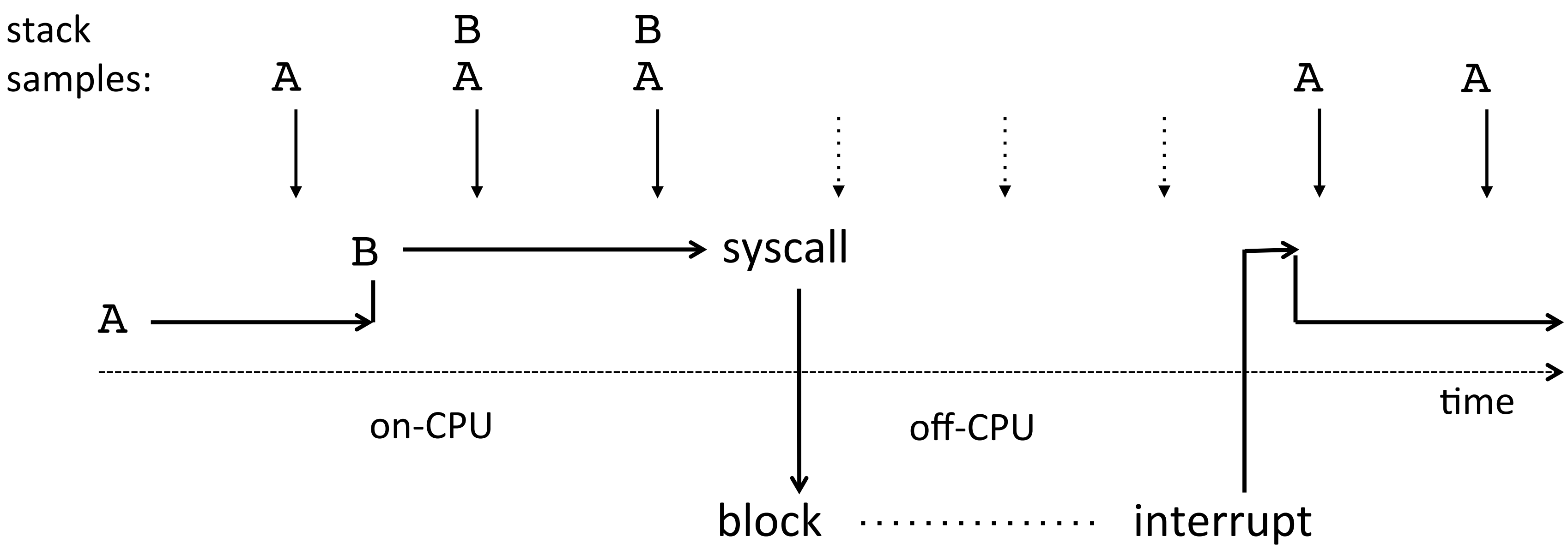
Performance of parallel applications

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 becomes parallel

CPU profiling (using sampling)

- Record stacks at a timed interval: simple and effective
  - Pros: Low (deterministic) overhead
  - Cons: Coarse accuracy, but usually sufficient



Performance of parallel applications

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

- **Polling (sampling)**
  - the application is periodically interrupted to collect performance data
  - Example: `gprof`

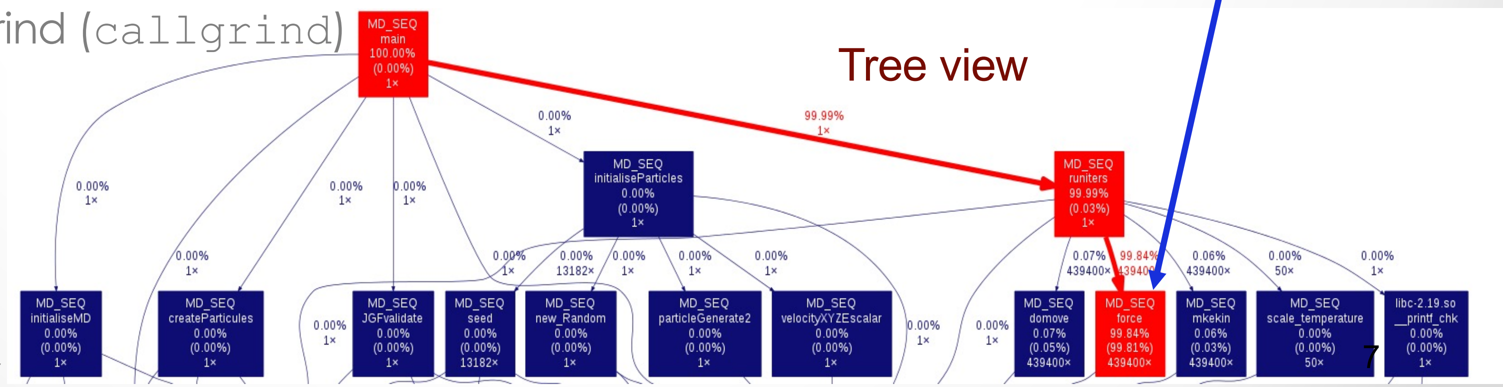
Flat view

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
50.00	2.47	2.47	2	1.24	1.24	matSqrt
24.70	3.69	1.22	1	1.22	1.22	matCube
24.70	4.91	1.22	1	1.22	1.22	sysCube
0.61	4.94	0.03	1	0.03	4.94	main
0.00	4.94	0.00	2	0.00	0.00	vecSqrt
0.00	4.94	0.00	1	0.00	1.24	sysSqrt
0.00	4.94	0.00	1	0.00	0.00	vecCube

Hot-spot

- **Instrumentation**
  - programmer (or tools) introduce code to collect performance data on useful events
    - may produce better results but also produces more interference (e.g., overhead)
  - Example: Valgrind (`callgrind`)

Tree view

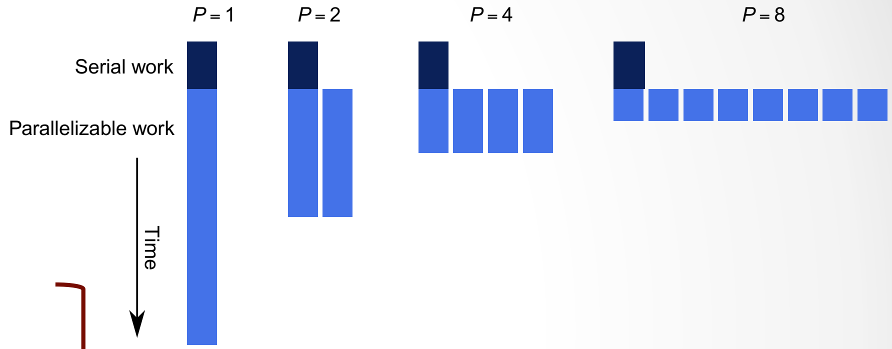


Scalability problems in shared memory

Why parallel applications do not have an ideal speedup?

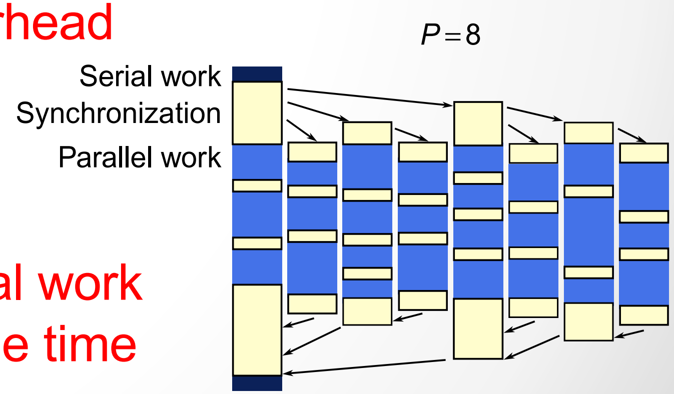
1. % of serial work (Amdahl's law)
2. Memory wall
   - Serializes memory accesses

Serial work



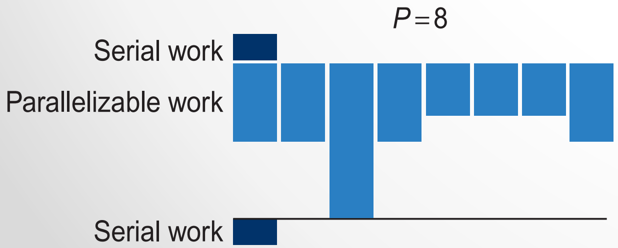
3. Parallelism/task granularity
   - Additional work on the parallel application (tasks management, redundant computations, …)
4. Synchronisation overhead
   - Might also serialize execution (e.g. critical)
   - Includes (serial) calls to external routines (e.g., malloc)

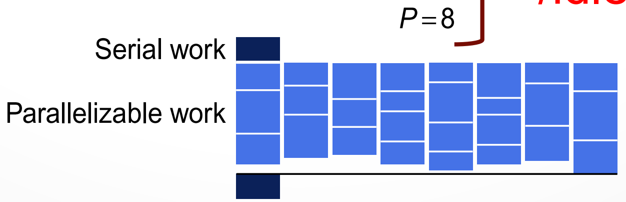
Parallelism overhead



5. Load imbalance
   - Over-decomposition can improve load balancing

Serial work /idle time





Scalability problems in shared memory

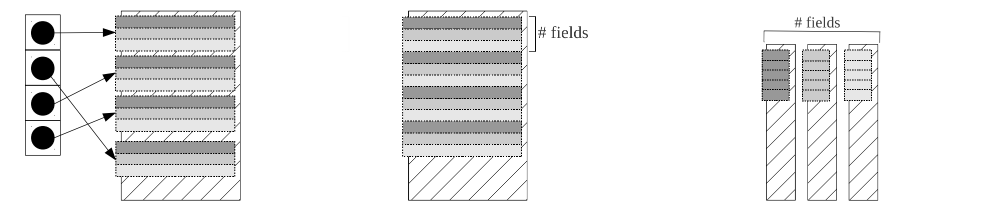
Some reasons for the lack of scalability *(1)*

2. Memory/cache bandwidth limitation

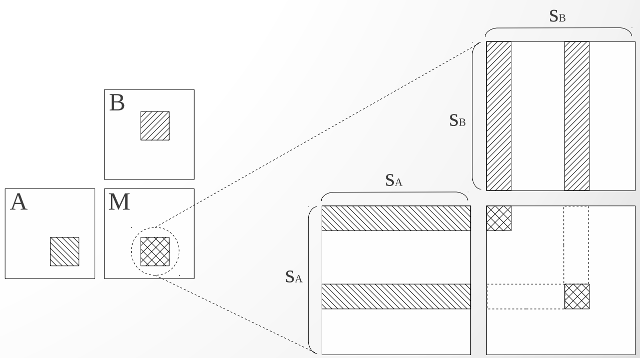
- Diagnostic (some options):
  1. Measure required mem bandwidth (per core) & compare against available bandwidth
  2. Computational intensity = #I / LLC.MISS (or L2.MISS)
  3. CPI increase with the number of threads
- Action:
  - Improve data locality
- Approaches

1) Convert AOP to AOS/SOA layout

2) Use loop tiling techniques



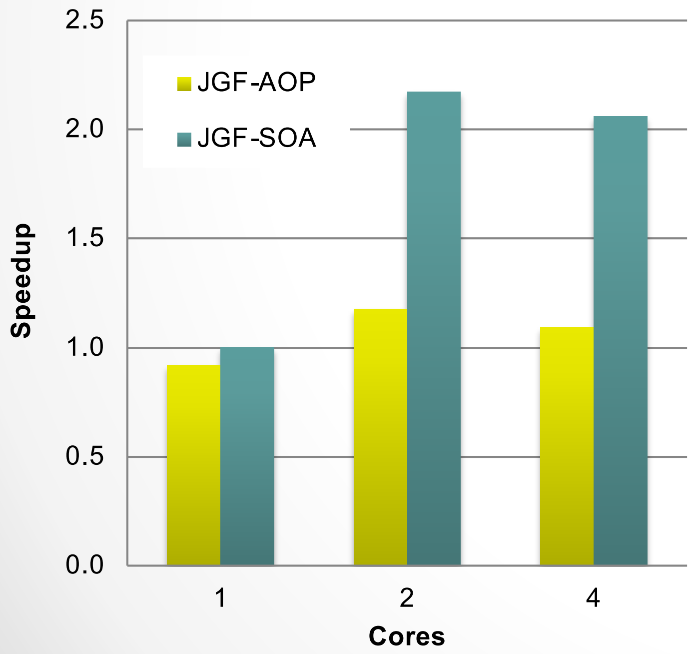
Array of Pointers (AoP) Array of Structures (AoS) Structure of Arrays (SoA)

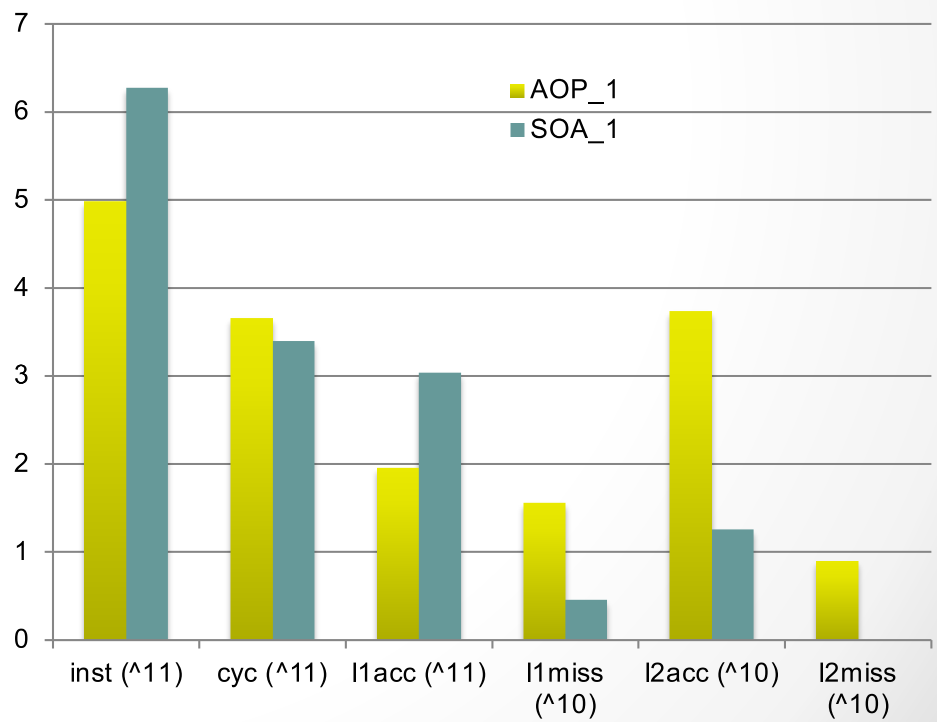


Scalability problems in shared memory

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 *(2)*

3. Fine-grained parallelism (excessive parallelism overhead)

- Diagnostic:
  - Measure task granularity (computation/parallelism ratio)
    (#I seq vs. sum #I par)
- Action:
  - Increase task size 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
- 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];
}
```

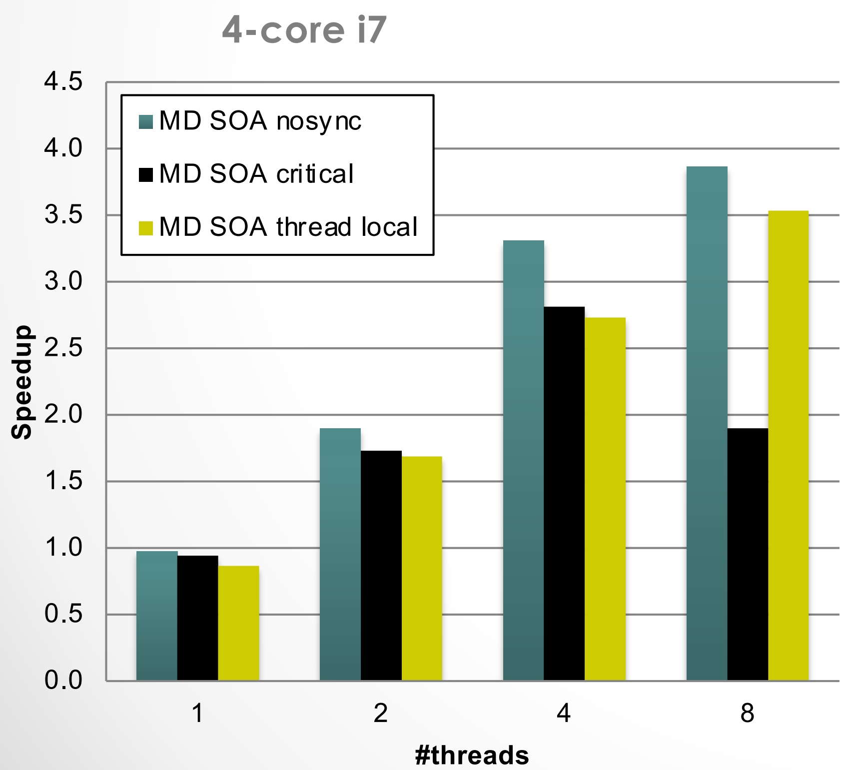
→

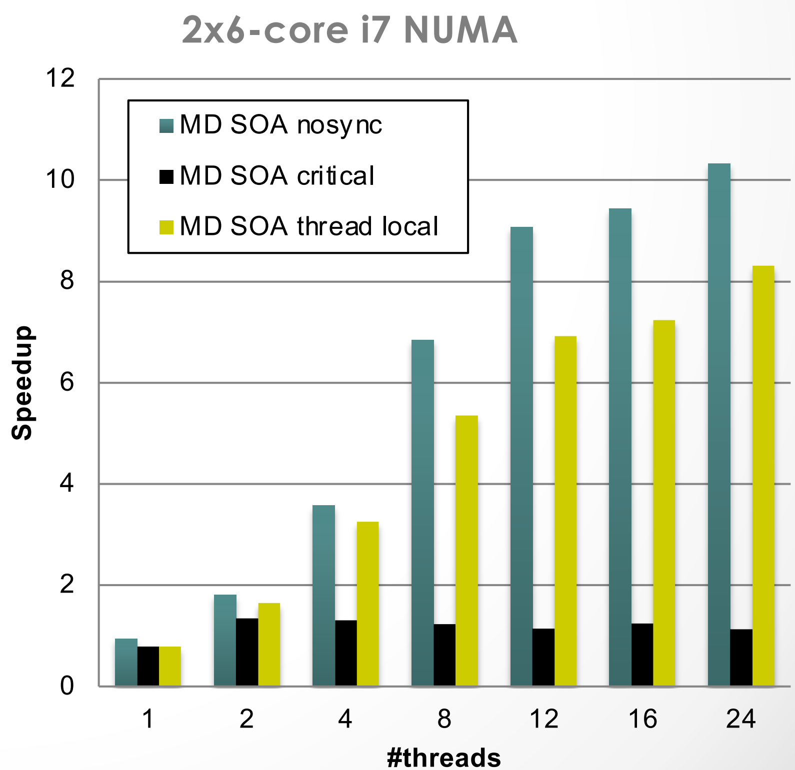
```
sum = 0;
# pragma omp parallel for reduction(+:sum)
for(int i = 0;  i<100; i++) {
          sum += array[i];
}
```

Scalability problems in shared memory

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

- **Example:** excessive task synchronisation
  - Scalability of JGF MD





Scalability problems in shared memory

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

4. Poor load distribution (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++) {
        …
}
```

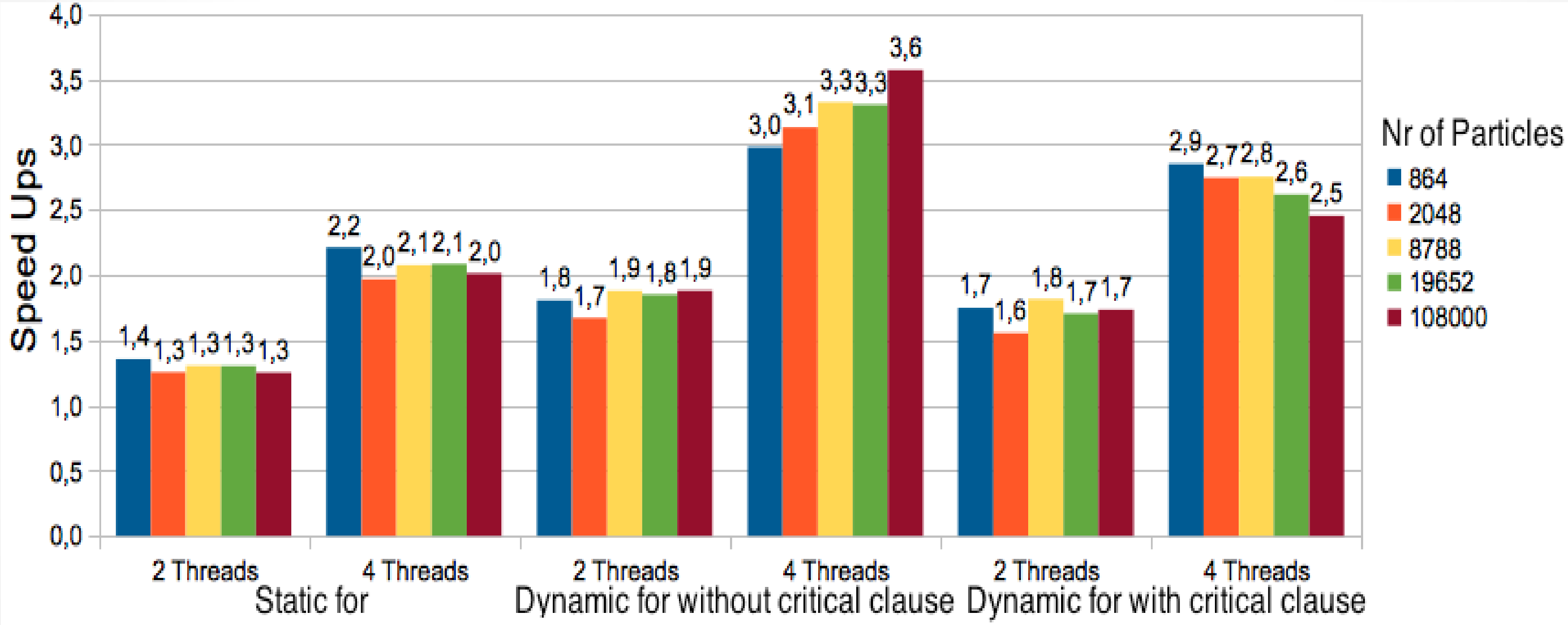
→

```
# 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

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

- **Example:** MD load distribution



Scalability problems in shared memory

Summary:

Possible metrics to present

1. **% of serial work**

2. **Memory bandwidth and computational intensity**
   - locality optimisations

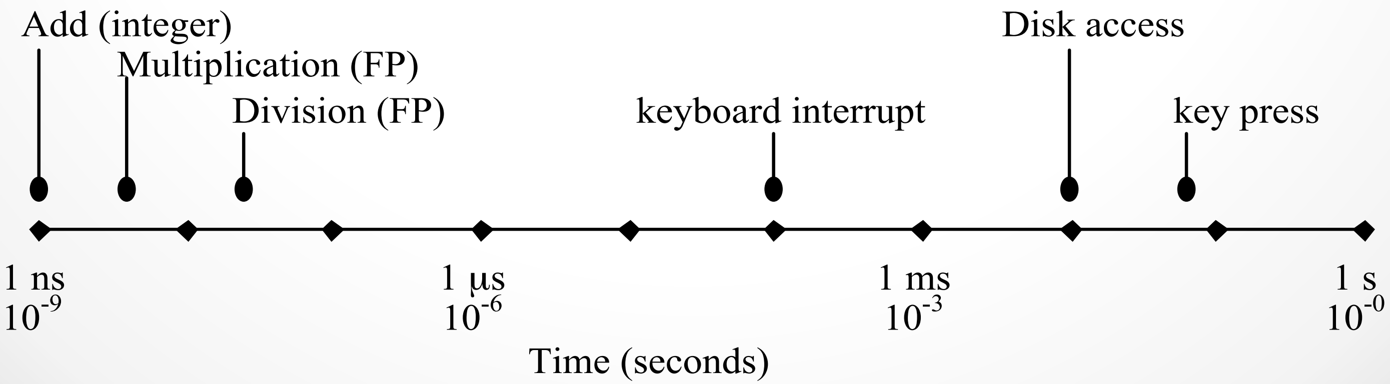
3. **Task granularity / parallelism overhead**
   - increase granularity

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

5. **Compute time per parallel task**

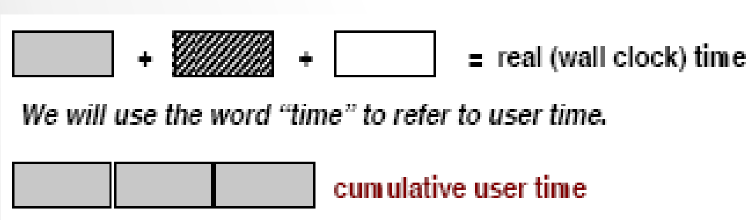
Measuring performance

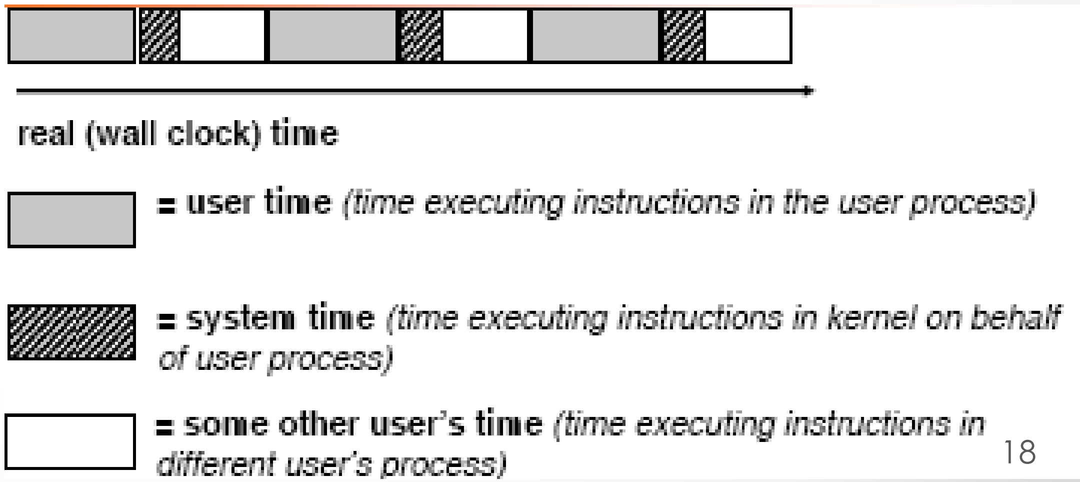
- **Principles**
  - **Isolate from external factors**
    - Consider the measurement overhead
    - Repeat the measurement
    - Avoid other system load
  - **Document the experiment to be reproducible by others**
    - Hardware, software versions, system state…
  - **Important: clock resolution**
    - <u>Precision</u>: difference between measured and real time
    - <u>Resolution</u>: time unit between clock increments
      - In principle, 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 the end of execution
    - Depends on the system load, I/O, …
  - **Complexities**
    - Process scheduling (10ms?)
    - Load introduced by other processes (e.g., garbage collector in JVM, other users)





Measuring performance

- **Options to measure execution time**

  - **"`time`" command line**
    - Only for measurements >>1seg

```
[jls@compute-652-2]$ time ./a.out
F=102334155 Time=0.935394

real        0m0.938s
user        0m0.934s
sys         0m0.001s
```

  - **`gettimeofday()`**
    - Returns the number of microseconds since 1-Jan-1970
    - Uses the "Timer" or the cycle counter (depends on the platform)
    - Best case resolution: 1μs
  - **Clock cycle counter** (introduced in modern processors)
    - High resolution
    - Useful for measurements <<1s
  - **Timer function in OpenMP / MPI**
    - *`omp_get_wtime`*, *`omp_get_wtick`*
    - *`MPI_Wtime`*
  - **System.nanoTime()** (in Java 4)

High resolution implementations!

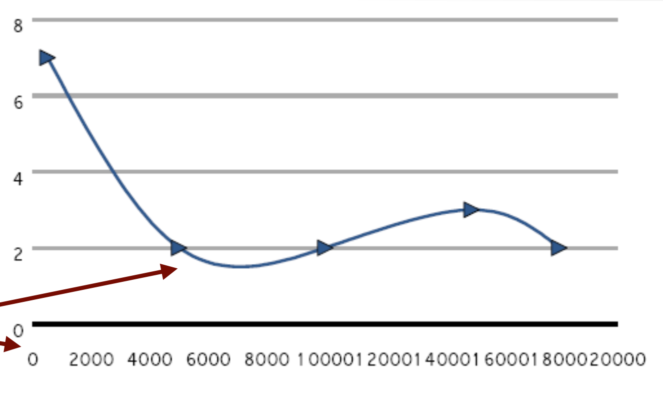
Measuring performance

- **How to combine results from several measurements?**
  - **Average**
    - Affected by extreme high/low values
    - Additionally: show the deviation among measurements *(standard deviation)*
  - **Best measure**
    - Value in ideal conditions
  - **Average of k-best**
    - k-best within a given tolerance
    - Best guarantee for reproducible results
    - Removes outliers
  - **Median**
    - More robust to large variations

Measuring performance

- **Presenting results**
  - ➢ Present results in a readable (& 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: **1,00~~004~~ 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)