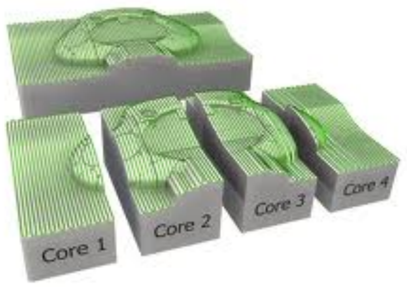
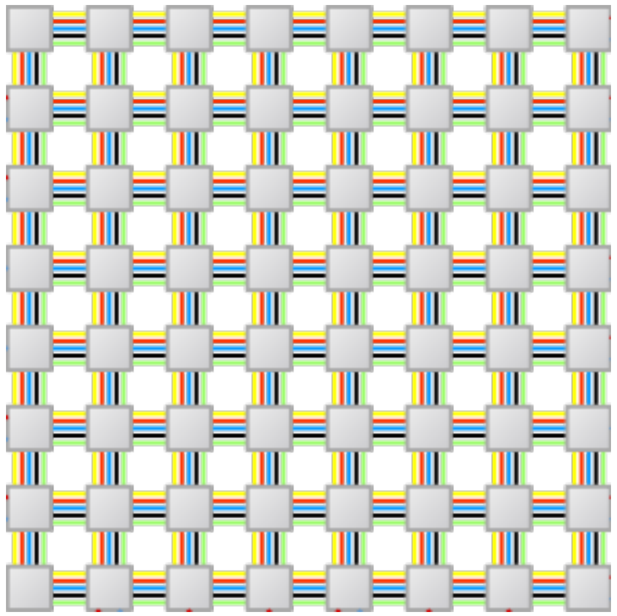
Scalability on Manycore Machines

Sequential
84 seconds





Got
2 seconds

Expected
84/84 = 1 second

!?!

Got
25 seconds

Claude TADONKI

MINES ParisTech – PSL Research University
Centre de Recherche Informatique
claude.tadonki@mines-paristech.fr

GDR-ISIS Seminar
LIP6 at UPMC
April 25, 2017, Paris (France)

MINES ParisTech

PSL RESEARCH UNIVERSITY PARIS

Conceptual key factors related to scalability

Code to be parallelized

☞ Loss of parallel efficiency !!!!

Speedup $\sigma(p) = T_s / T_p$

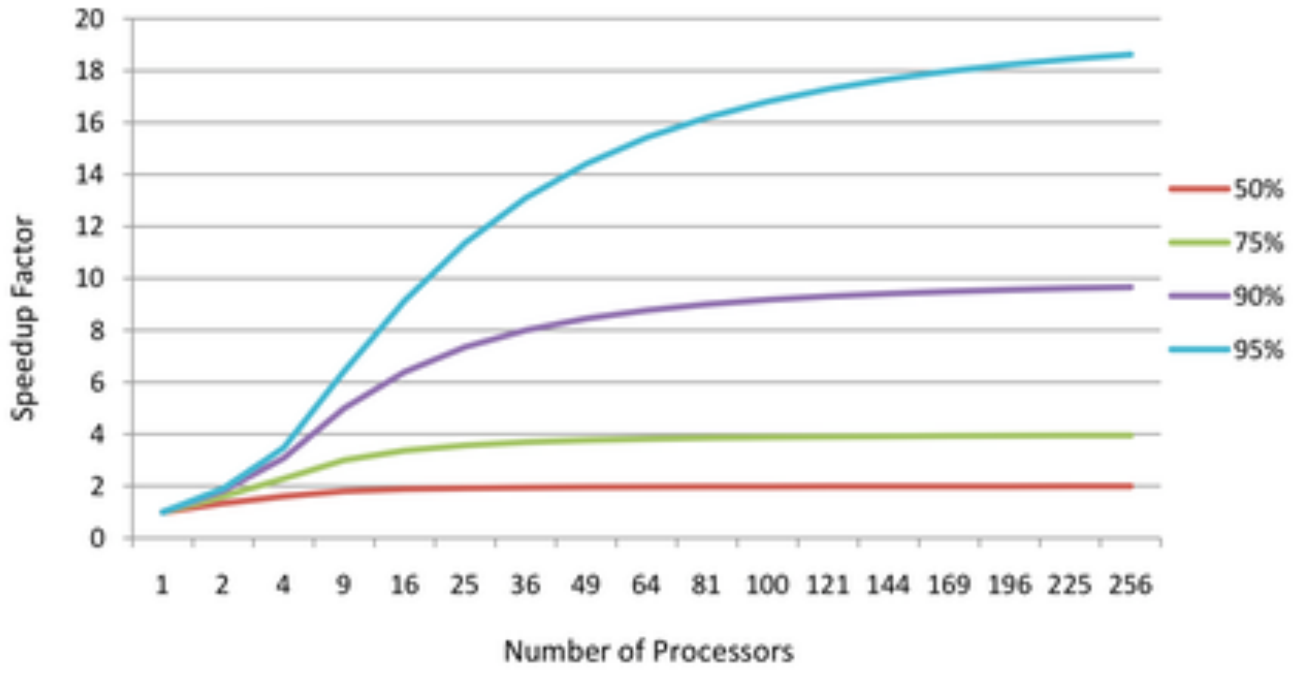
Efficiency (parallel) $e = \sigma(p) / p$

- Always keep in mind that these metrics only refer to "how go is our parallelization".
- They normally quantify the "noisy part" of our parallelization.
- A good speedup might just come from an inefficient sequential code, so do not be so happy !
- Optimizing the reference code makes it harder to get nice speedups.
- We should also parallelize the "noisy part" so as to share its cost among many CPUs.

Amdahl's Law illustration

p	par = 95%	par = 90%	par = 75%	par = 50%
1	100.00	100.00	100.00	100.00
2	52.50	55.00	62.50	75.00
4	28.75	32.50	43.75	62.50
8	16.88	21.25	34.38	56.25
16	10.94	15.62	29.69	53.12
32	7.97	12.81	27.34	51.56
64	6.48	11.41	26.17	50.78
128	5.74	10.70	25.59	50.39
256	5.37	10.35	25.29	50.20
512	5.19	10.18	25.15	50.10

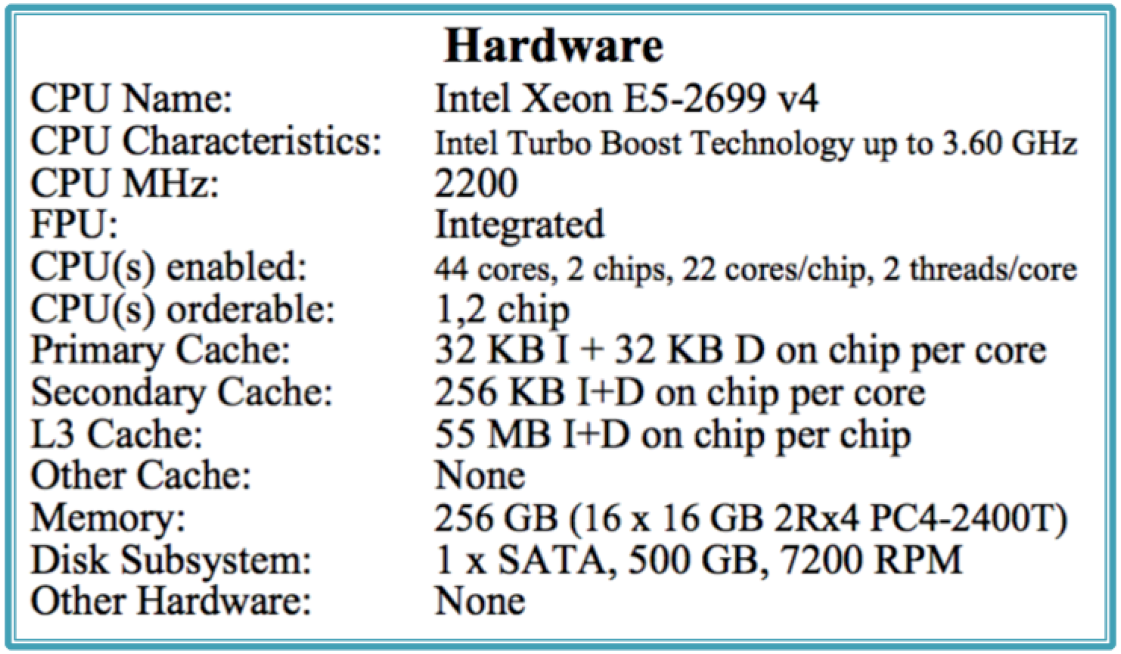
Simulated parallel timings



INTEL BROADWELL

Intel® Xeon® Processor E5-2699 v4
Released in April 2016

- 22x2 = 44 cores
- 2.2 Ghz/core
- 3.6 GHz Boost
- Hyperthreading
- 256-bit vectors
- 256 Gb RAM
- 76.8 Gb/s
- 500 Gb disk
- 1.54 Tflops SP
- 0.78 Tflops DP
- **Tflops** is **1000 000 000 000** (1 billion) floating point **operations per seconds**



Hardware

CPU Name:	Intel Xeon E5-2699 v4
CPU Characteristics:	Intel Turbo Boost Technology up to 3.60 GHz
CPU MHz:	2200
FPU:	Integrated
CPU(s) enabled:	44 cores, 2 chips, 22 cores/chip, 2 threads/core
CPU(s) orderable:	1,2 chip
Primary Cache:	32 KB I + 32 KB D on chip per core
Secondary Cache:	256 KB I+D on chip per core
L3 Cache:	55 MB I+D on chip per chip
Other Cache:	None
Memory:	256 GB (16 x 16 GB 2Rx4 PC4-2400T)
Disk Subsystem:	1 x SATA, 500 GB, 7200 RPM
Other Hardware:	None

LQCD performance on a 44 cores processor

$$D\psi(x) = A\psi(x) - \frac{1}{2}\sum_{\mu=0}^{4}\{[(I_4 - \gamma_\mu) \otimes U_{x,\mu}]\psi(x+\hat{\mu}) + [(I_4 + \gamma_\mu) \otimes U^{\dagger}_{x-\hat{\mu},\mu}]\psi(x-\hat{\mu})\}$$

#cores	#threads	t(s)	GFlops	Speedup
1	2	0.02552	9.98	1
2	4	0.01301	19.59	1.96
4	8	0.00679	37.50	3.76
8	16	0.00475	53.60	5.37

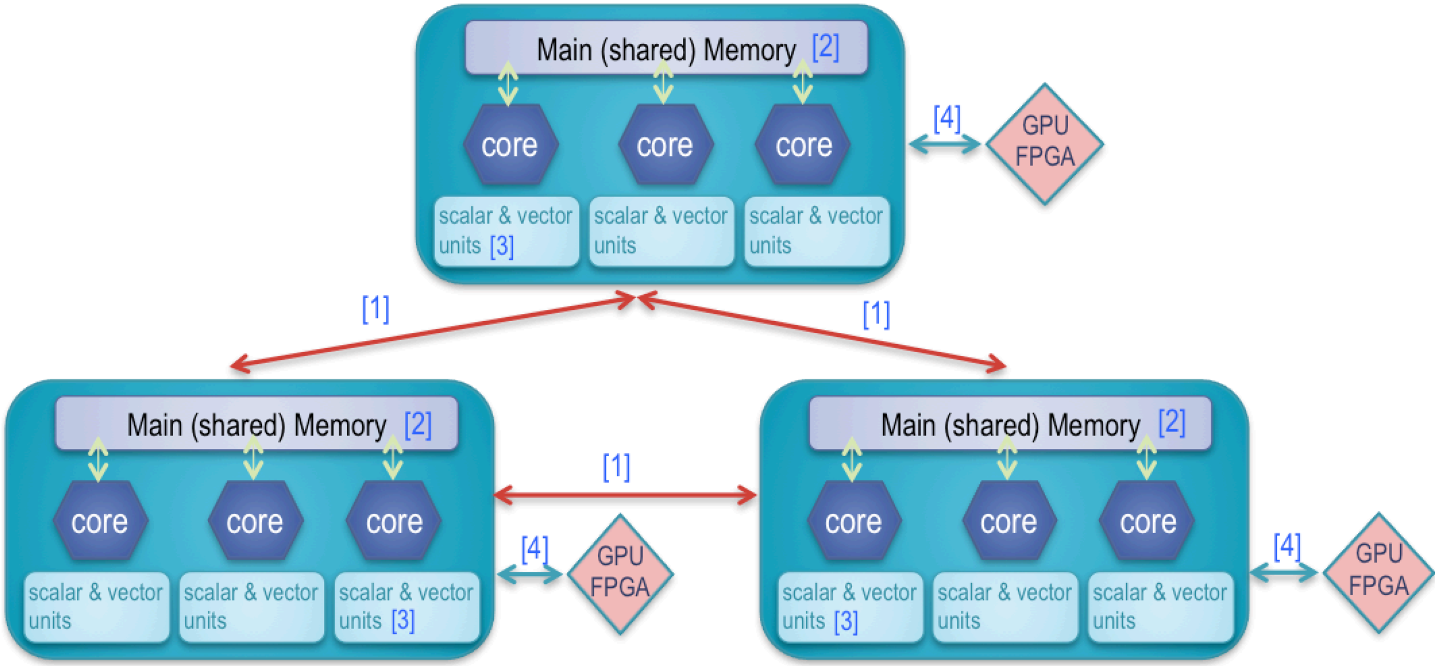
Optimal absolute performance on a single core and good scalability !!!

(2 nodes) 16	32	0.00476	53.53	5.36
(4 nodes) 32	64	0.00507	50.25	5.03

Something happened !!!

Let's now explore and understand it.

- **Speedup is just one component of the global efficiency**
- **We need to exploit all levels of parallelism in order to get the maximum SC performance**
  - Message passing between nodes (MPI, ...) [1]
  - Shared memory between cores (Pthreads, OpenMP, ...) [2]
  - Vector computing inside a core (SSE,AVX, ...) [3]
  - Accelerated computing beside a node (Cuda, OpenCL, ...) [4]



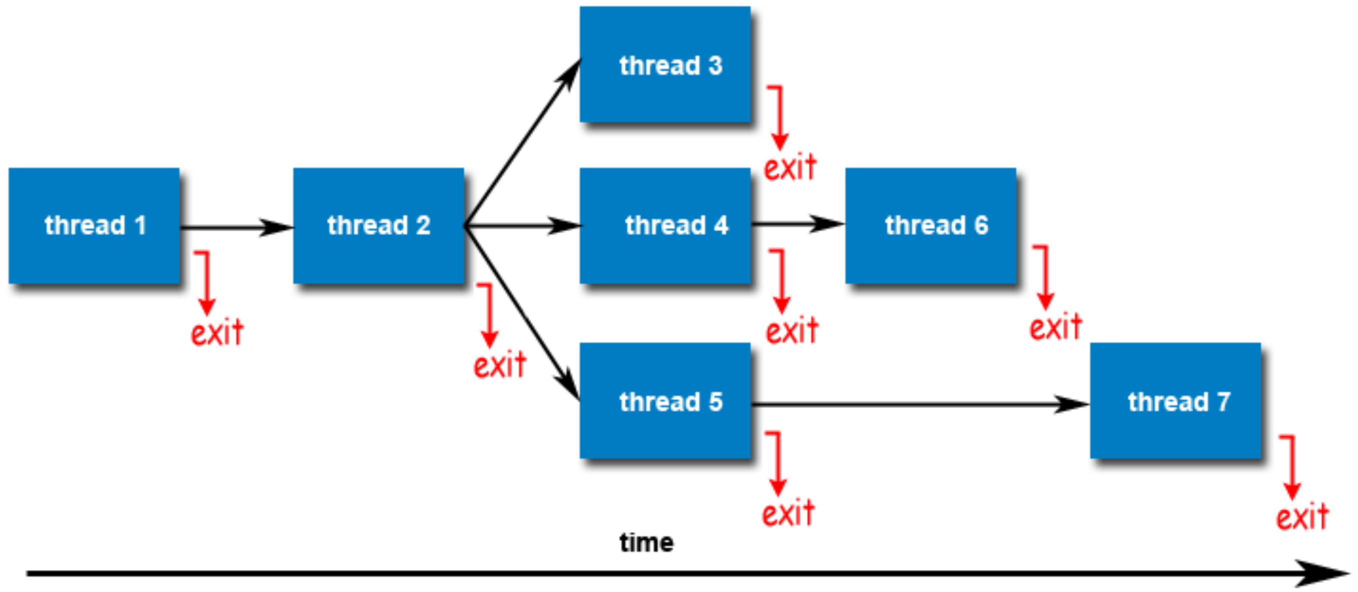
- **Because of cost from explicit interprocessor communication, a scalable SMP implementation on a (manycore) compute node is a rewarding effort anyway.**

Main factors against scalability on a shared memory configuration

- **Threads creation and scheduling**
- **Load imbalance**
- **Explicit mutual exclusion**
- **Synchronization**
- **Overheads of memory mechanisms**
  - Misalignment (when splitting arrays)
  - False sharing
  - Bus contention
  - NUMA effects

Let's now examine each of these aspects.

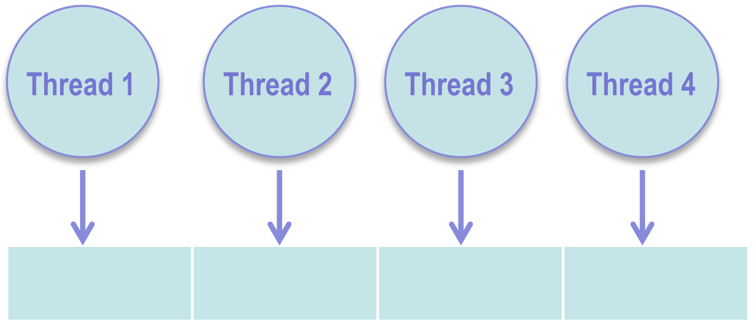
Thread creation and scheduling

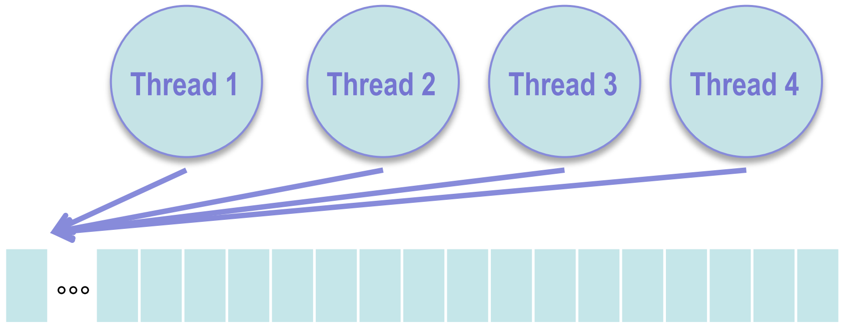


- **Thread creation + time-to-execution yield an overhead (usually marginal)**
  - Creating an pool of (always alive) threads that operate upon request is one solution
- **Dynamic threads migration could break some good scheduling strategies**
- **Threads allocation without any affinity could result in an inefficient scheduling**
- **The system might consider only part of available CPU cores**
- **Threads scheduling regardless of conceptual priorities could be inefficient**

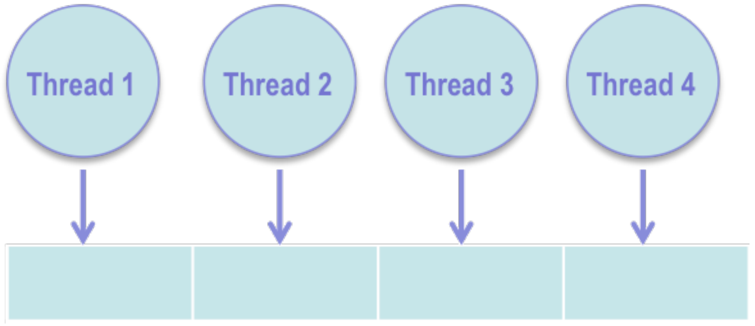
Load imbalance or unequal execution times

- **Tasks are usually distributed from static-based hypotheses**
- **Effective execution time is not always proportional to static complexity**
- **Accesses to shared resources and variables will incur unequal delays**
- **The execution time of a task might depend on the values of the inputs or parameters**
  - Influence on the execution path following the controls flow
  - Influence on the behavior because of numerical reasons
  - Constraints overhead from particular data location
  - Specific nature of data from particular instances (sparse, sorted, combinatorial complexity, …)
- **We thus need to seriously consider the choice between static and dynamic allocations**





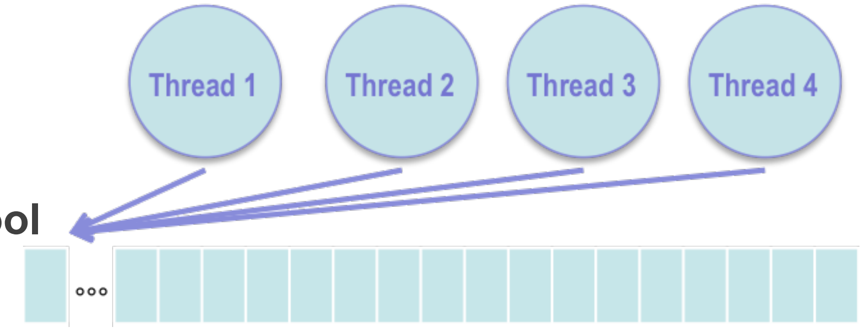
Static block allocation



block, cyclic or block-cyclic

- This is the most common allocation
- Each thread is assigned a predetermined block
- Assignment can be from input or output standpoint
- 🟢 The need for synchronization is unlikely
- 🔴 Equal chunks do no imply equal loads

Dynamic allocation with a pool of tasks

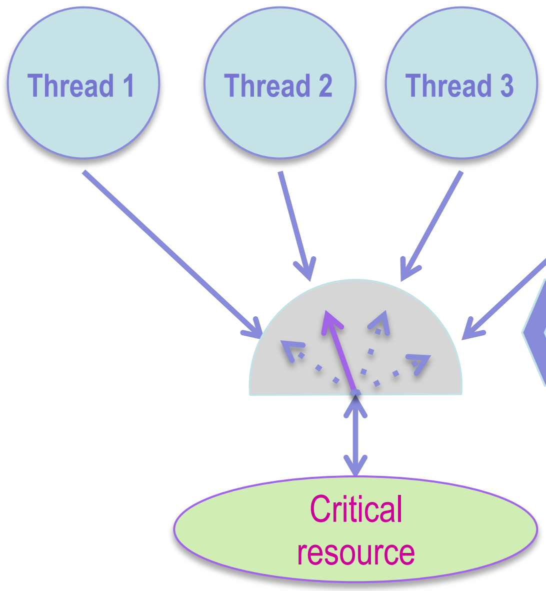


The granularity is important

- Increasingly considered
- Thread continuously pop up tasks from the pool
- Usually organized from output standpoint
- 🟢 **More balanced completion times are expected** (effective load balance)
- 🔴 **Synchronization is needed to manage the pool** (some overhead is expected)

The choice depends on the nature of the computation and the influence of data accesses

Explicit mutual exclusion

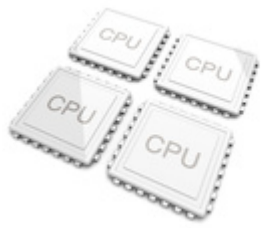


Several threads are asking for the critical resource and typically get locked

Only **one thread is selected** to get the critical resource and the others remain locked

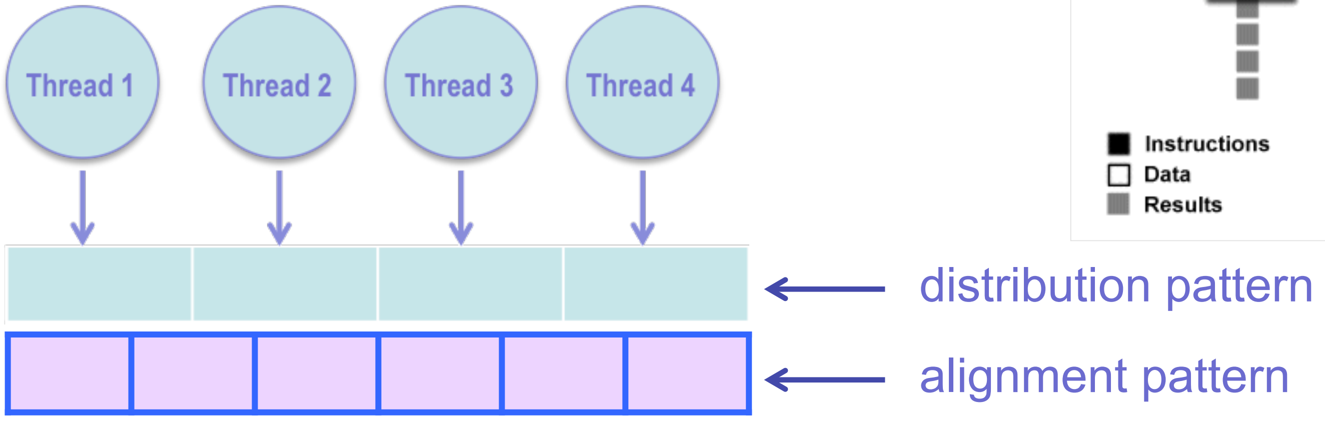
Critical resource is thus given to the requesting threads on a **purely sequential basis**

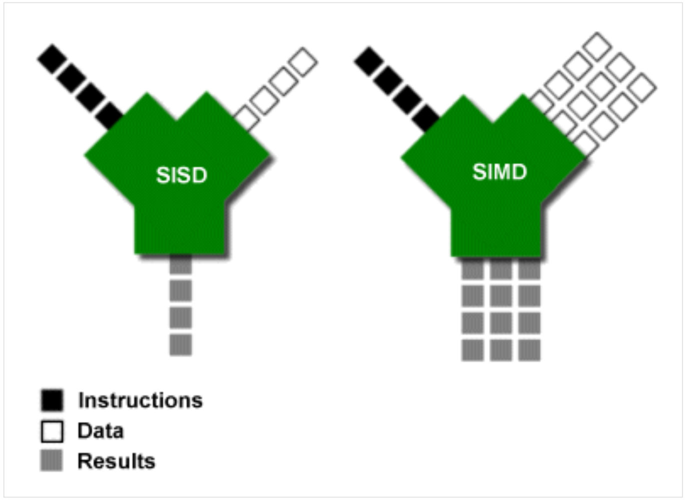
- **Applies on critical resources sharing**
- **Applies on objects that cannot/should be accessed concurrently** (file, single license lib, …)
- **Used to manage concurrent write accesses to a common variable**
- ● **A <u>non selected</u> thread can choose to <u>postpone its action</u> and <u>avoid being locked</u>**
- ● **Since this yields a sequential phase, it should be used skilfully** (only among the threads that share the same critical resource – strictly restricted to the relevant section of the program)



Since memory is (seamlessly) shared by all the CPU cores in a multicore processor, the overhead incurred by all relevant mechanisms should be seriously considered.

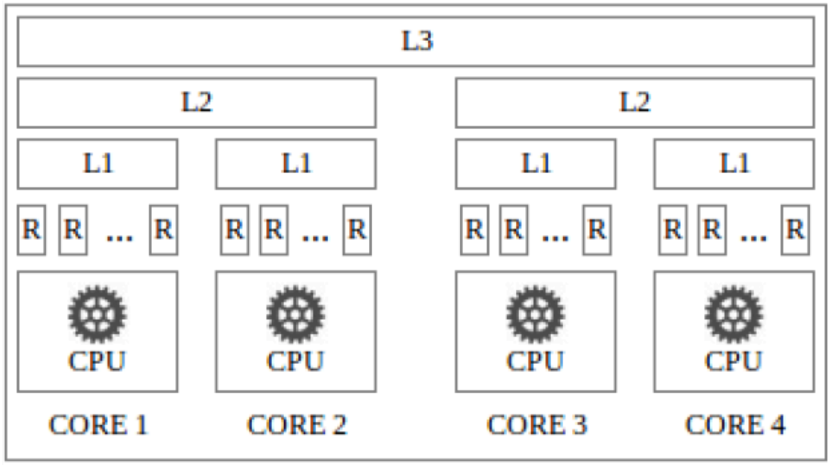
- **In case of a direct block distribution, some threads might received unaligned blocks.**



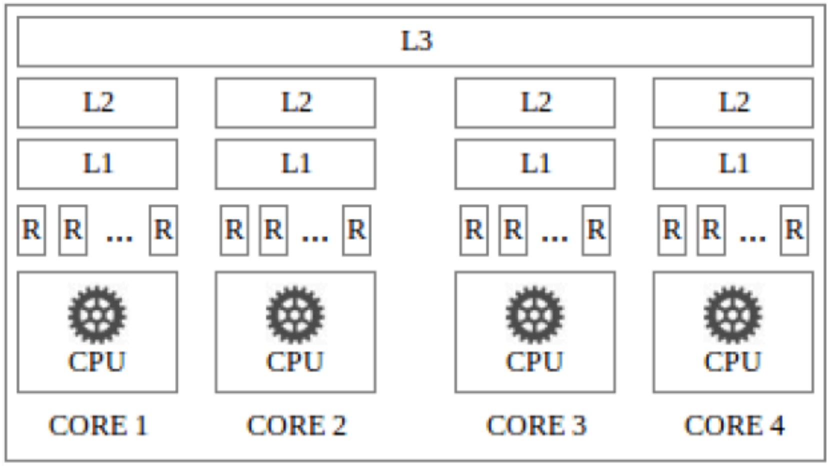


- **Threads to whom unaligned blocks are assigned will experience a slowdown**
- **The impact of misalignment is particularly severe with vector computing**
- **Always keep this in mind when choosing the number of threads and splitting arrays**

MEMORY: Levels of cache



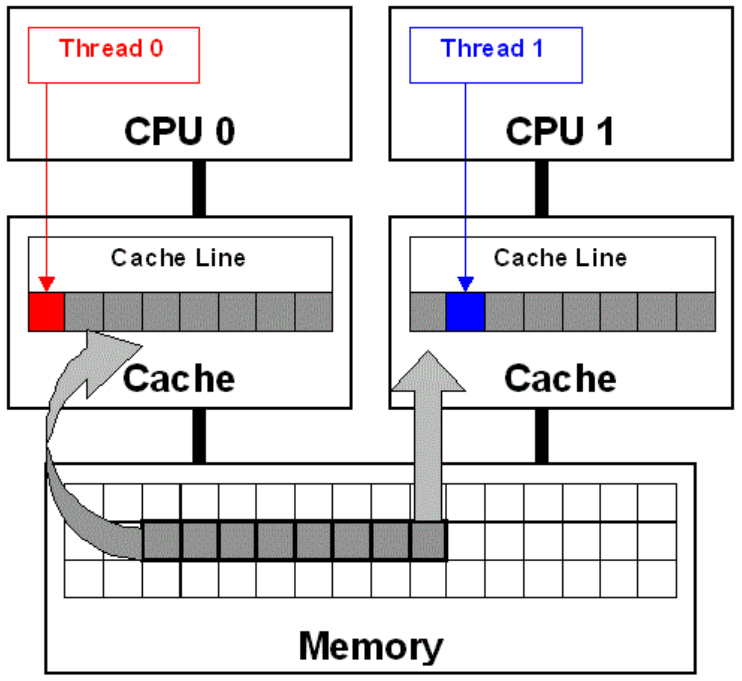
(a) Shared L2 cache



(b) Private L2 cache

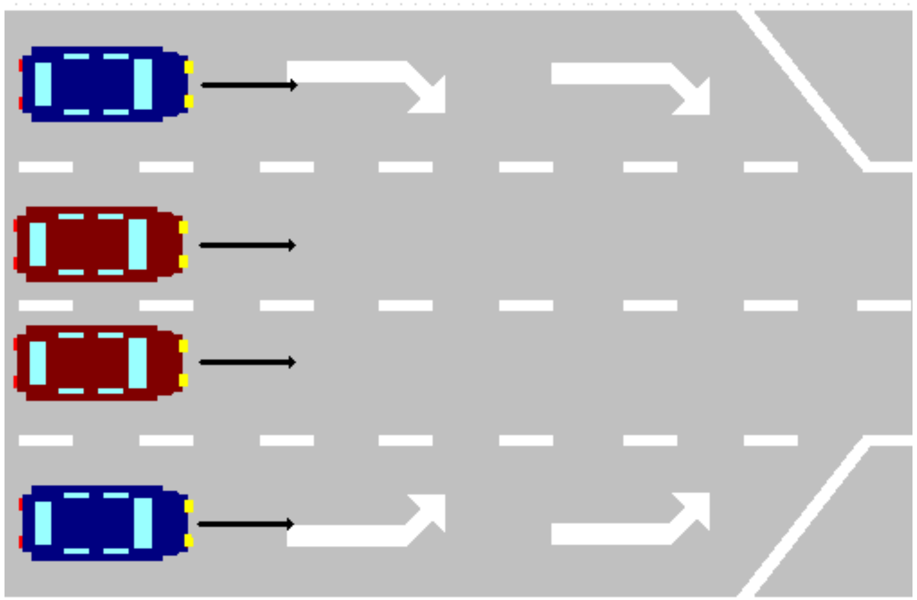
- The organization of the memory hierarchy is also important for memory efficiency
- **Case (a):**
  Assigning two threads which share lot of input data to C1 and C3 is inefficient
- **Case (b):**
  In place computation will incur a noticeable overhead due to coherency management
- Frequent thread migrations can also yield loss of cache benefit
- We should care about memory organization and cache protocol

MEMORY: False sharing



- This the systematic invalidation of a duplicated cache line on every write access
- The conceptual impact of this mechanism depends on the cache protocol
- The magnitude of its effect depends on the level of cache line duplications
- A particular attention should be paid with in place computation

MEMORY: Bus contention



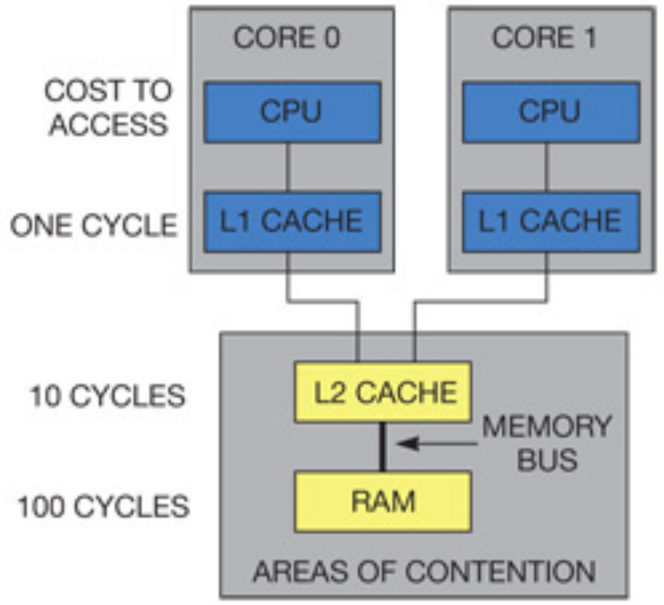
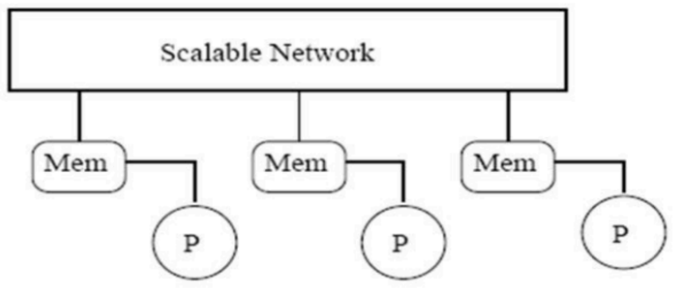


Figure 1 The latency to memory increases as you move up the hierarchy.

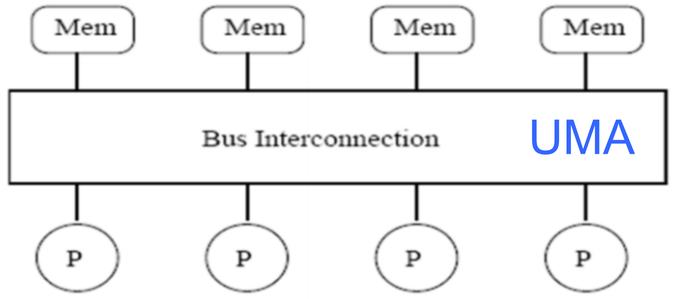
- The paths from L1 caches to the main memory fuse at some point (memory bus)
- As the number of threads is increasing, the contention is likely to get worse
- Techniques for cache optimization can help has they reduce accesses to main memory
- Redundant computation or on-the-fly reconstruction of data are worth considering

NUMA = **N**on **U**niform **M**emory **A**ccess

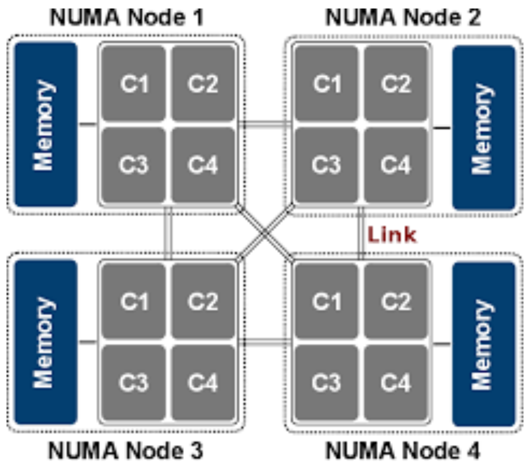
Shared Memory Architecture – NUMA

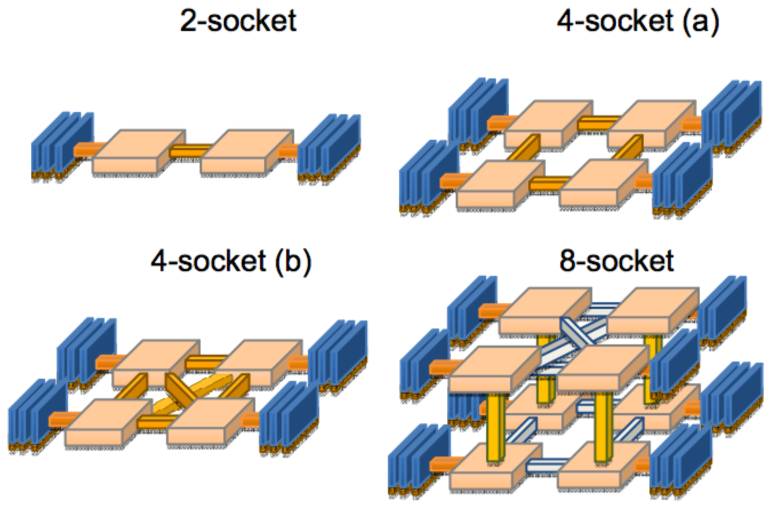


≠



- **The whole memory is physically partitioned but is still shared between all CPU cores**
- **This partitioning is seamless to ordinary programs as there is a unique addressing**
- **A typical configuration looks like this**

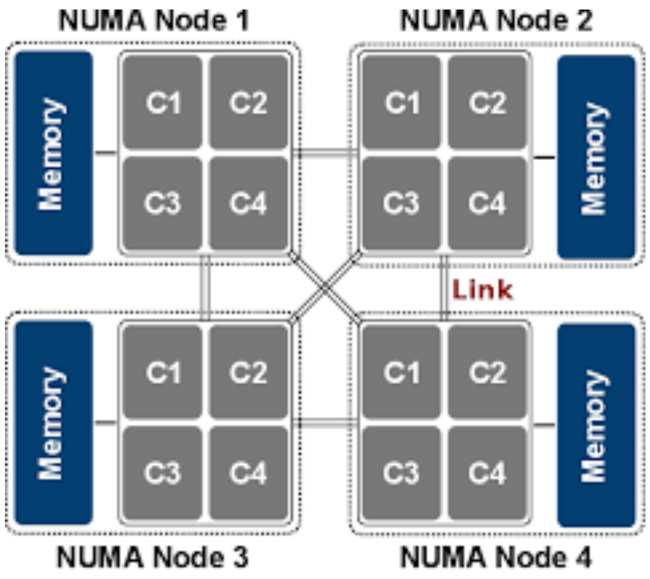




MEMORY: NUMA impacts

- **NUMA Nodes are linked by QPI links**
- **The distances matrix between NUM nodes is displayed by issuing** `numactl --hardware` **command**

```
node   0    1    2    3
  0:  10   11   21   21
  1:  11   10   21   21
  2:  21   21   10   11
  3:  21   21   11   10
```



- **These distances give an idea on how nodes are connected**
- **"Local accesses" are of course faster that "remote accesses"**
- **Links between NUMA nodes are potentially subject to heavy contention**
- **It is important to know the topology of the processor** (memory and CPU cores)
- **NUMA-unaware programs are likely to yield a noticeably poor scalability**
- **Memory allocation and thread binding to specific nodes are possible within programs**

MEMORY: NUMA management

- **NUMA considerations can be handled within programs through libraries like libnuma**

- **The library allow to**
  - **allocate memory on a specific node**
  - **ask to interleave an array on all NUMA nodes**
  - **check on which node a given memory space is allocated**
  - **identified on which NUMA node a given core** (logical id) **belongs to**

- **Such libraries should be used with flexibility in order to avoid portability issues**

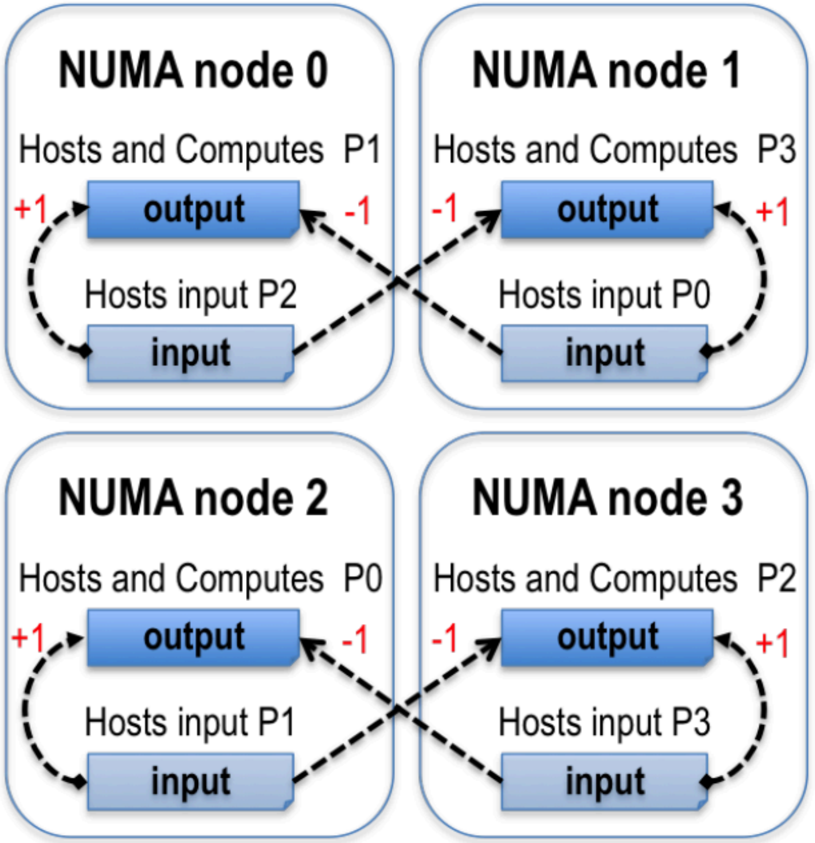
- **An efficient explicit management of NUMA considerations can improve scalability**

Successful NUMA Optimization (LQCD on Broadwell-EP)

$$D\psi(x) = A\psi(x) - \frac{1}{2}\sum_{\mu=0}^{4}\{[(I_4 - \gamma_\mu)\otimes U_{x,\mu}]\psi(x+\hat{\mu}) + [(I_4+\gamma_\mu)\otimes U^{\dagger}_{x-\hat{\mu},\mu}]\psi(x-\hat{\mu})\},$$

#cores	#threads	t(s)	GFlops	Speedup
1	2	0.02552	9.98	1
2	4	0.01301	19.59	1.96
4	8	0.00679	37.50	3.76
8	16	0.00475	53.60	5.37
(2 nodes) 16	32	0.00476	53.53	5.36
(4 nodes) 32	64	0.00507	50.25	5.03

#cores	#threads	t(s)	GFlops	Speedup
1	2	0.03025	8.42	1
2	4	0.01547	16.47	1.95
4	8	0.00825	30.87	3.66
8	16	0.00502	50.72	6.02
(2 nodes) 16	32	0.00305	83.65	9.33
(4 nodes) 32	64	0.00209	121.74	15.43



+1: dependencies $i + 1$ (modulo 4)
-1: dependencies $i - 1$ (modulo 4)

- Identify the main performance related characteristics of the processor
- Skilfully consider threads related features at programming level
- Design a NUMA-aware memory allocation and management strategy
- Consider preventing threads migration through thread binding statements
- Do your best to reduce accesses to main memory
- Address load imbalance or unequal thread completion times
- Use good profiling tools and proceed with incremental improvements

Thanks for your attention