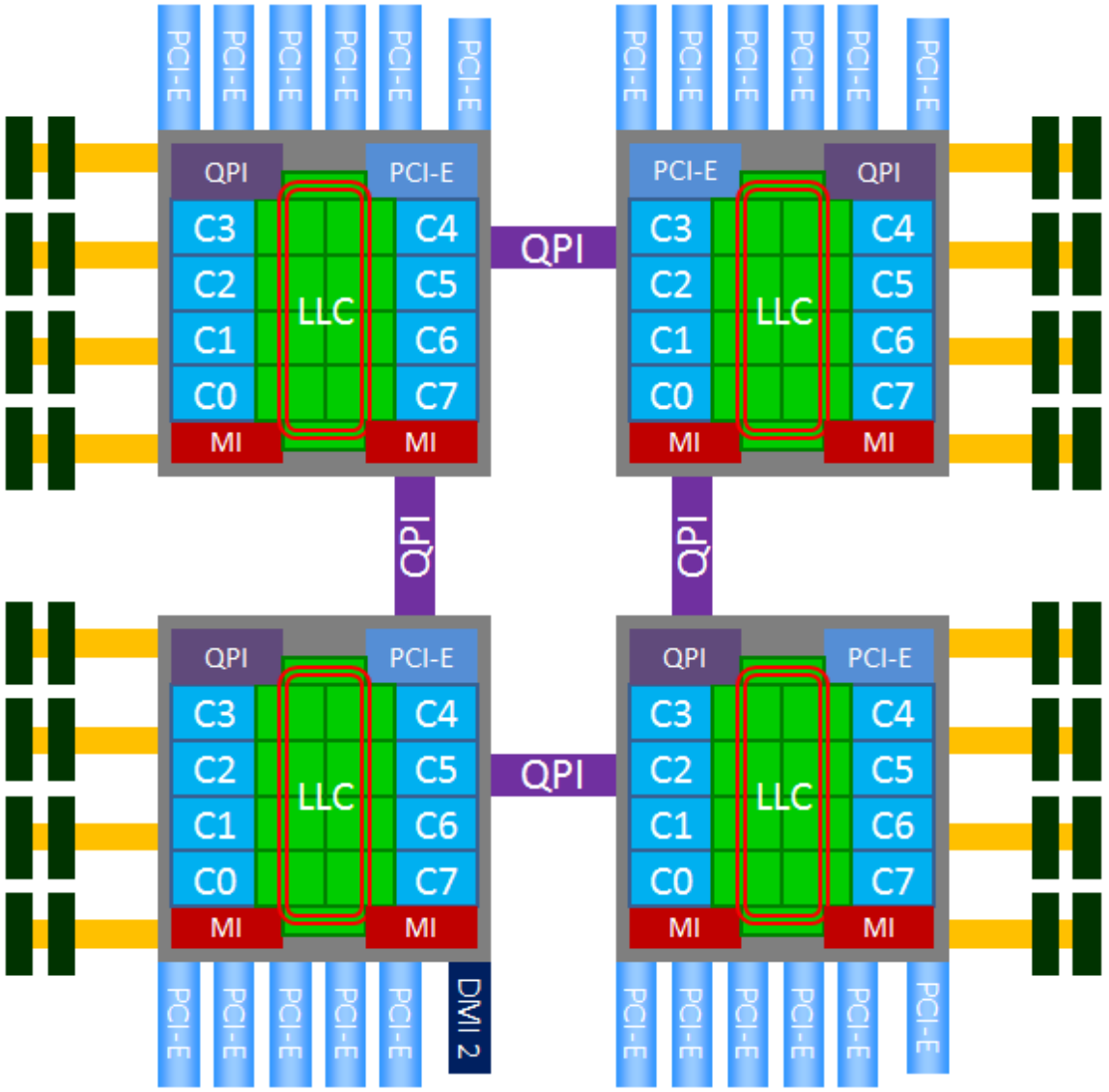
Analysis and modeling of

Computational Performance

Single node computational performance

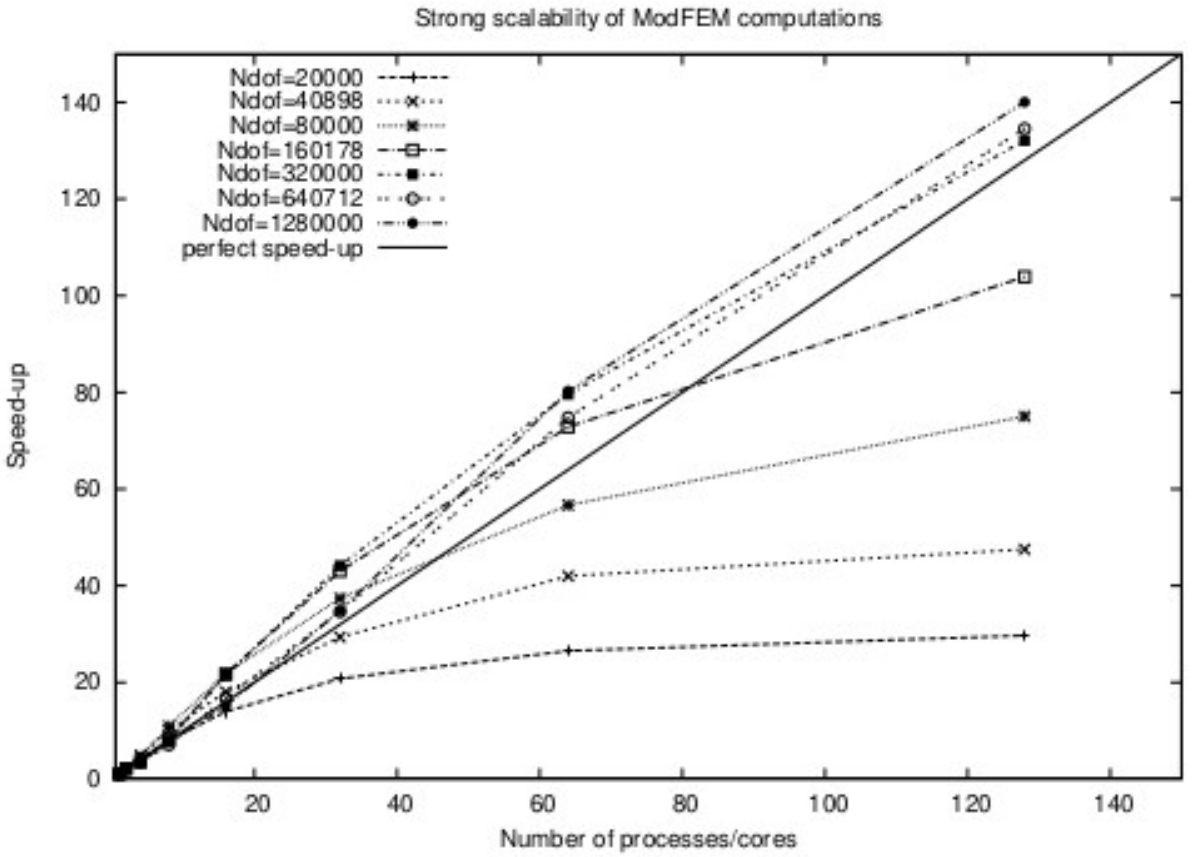
Single compute node



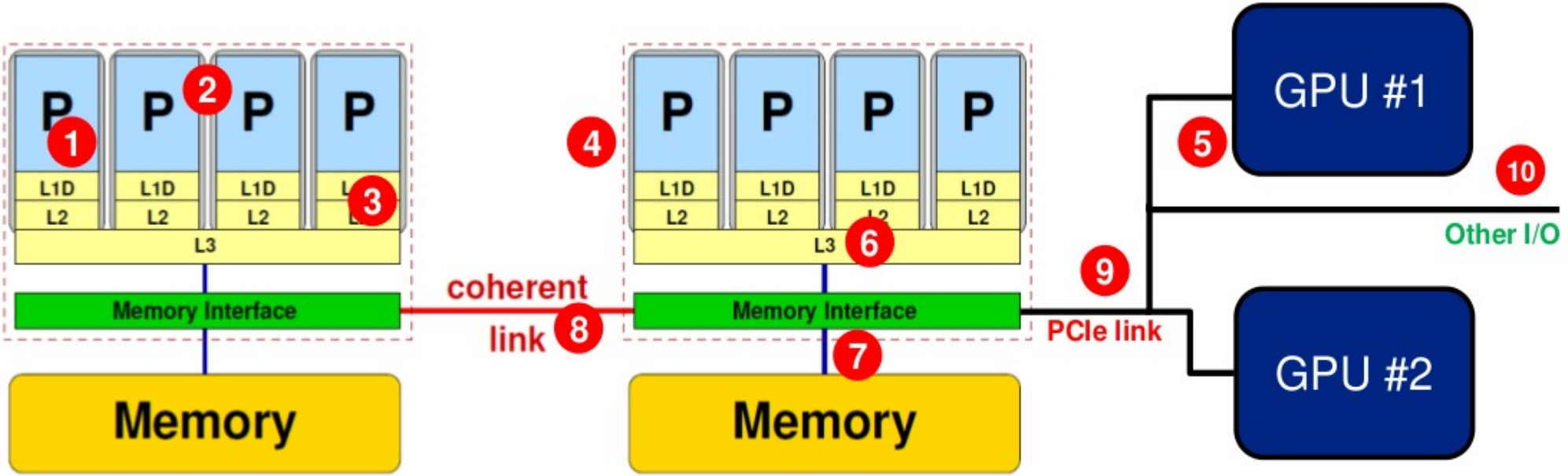
- Performance on a single compute node:
  - absolute
    - GFLOP/s
    - GB/s
  - relative
    - speed-up
    - efficiency
  - due to limited core and memory resources
    - speed-up is limited
    - strong scalability study is often performed for large problems (using most of DRAM memory)

Strong scalability

- Parallel speed-up
  - in theory
    - $S(p) = T_s / T_{||}(p)$
  - in practice
    - $S(p) = T_{||}(1) / T_{||}(p)$
- Parallel efficiency
  - $E(p) = 100\% * S(p) / p$
- Perfect speed-up
  - $S(p) = p$
  - $E(p) = 100\%$
- Superlinear speed-up is possible



Single compute node resources



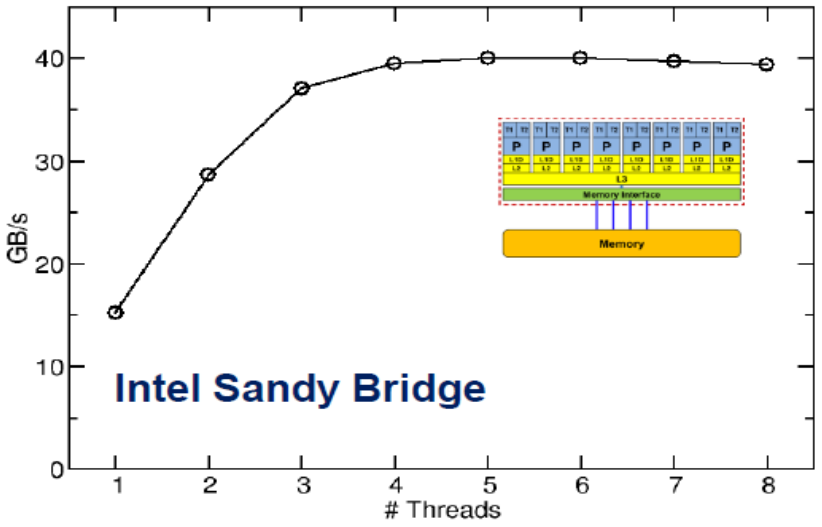
Parallel resources:

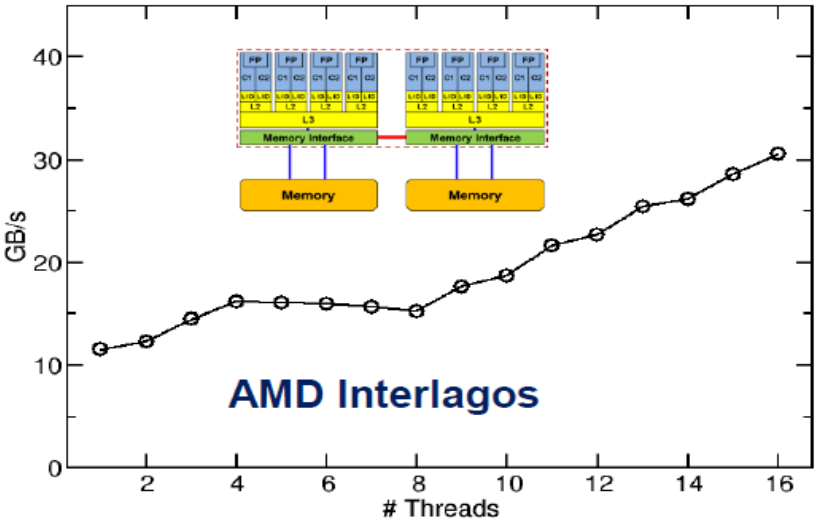
- Execution/SIMD units 1
- Cores 2
- Inner cache levels 3
- Sockets / ccNUMA domains 4
- Multiple accelerators 5

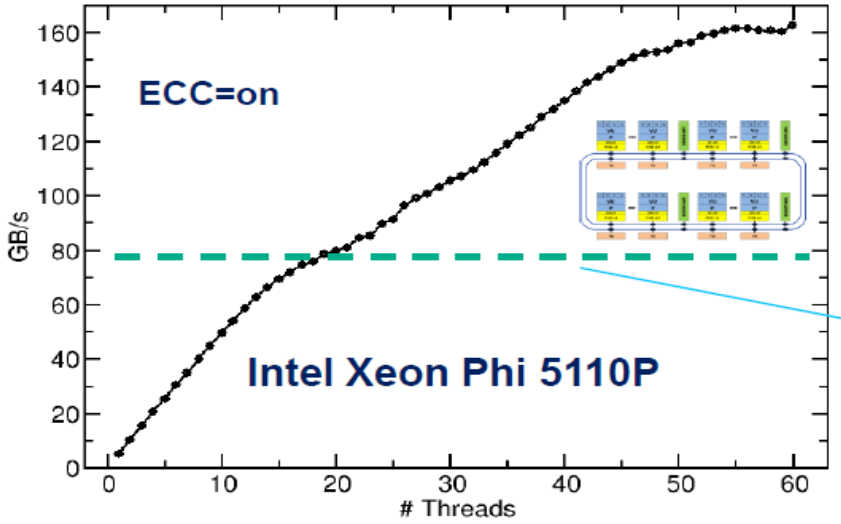
Shared resources:

- Outer cache level per socket 6
- Memory bus per socket 7
- Intersocket link 8
- PCIe bus(es) 9
- Other I/O resources 10

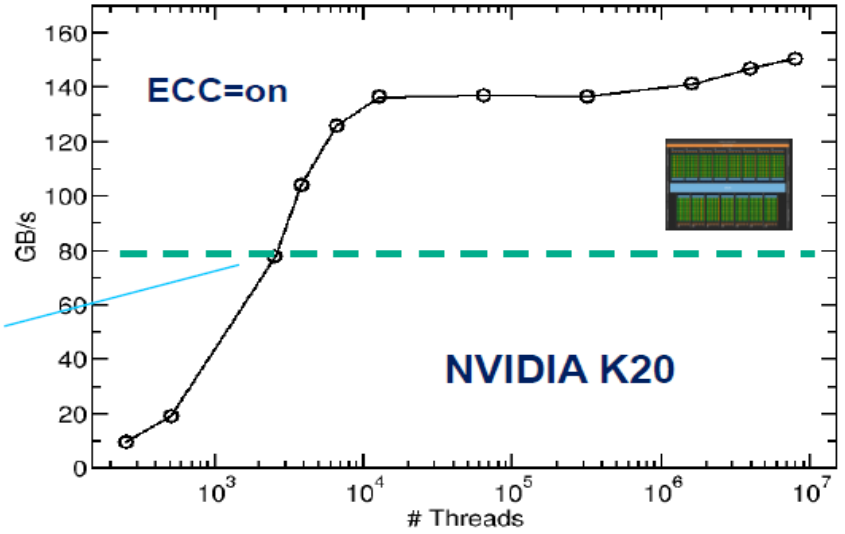
Memory scalability for different processors



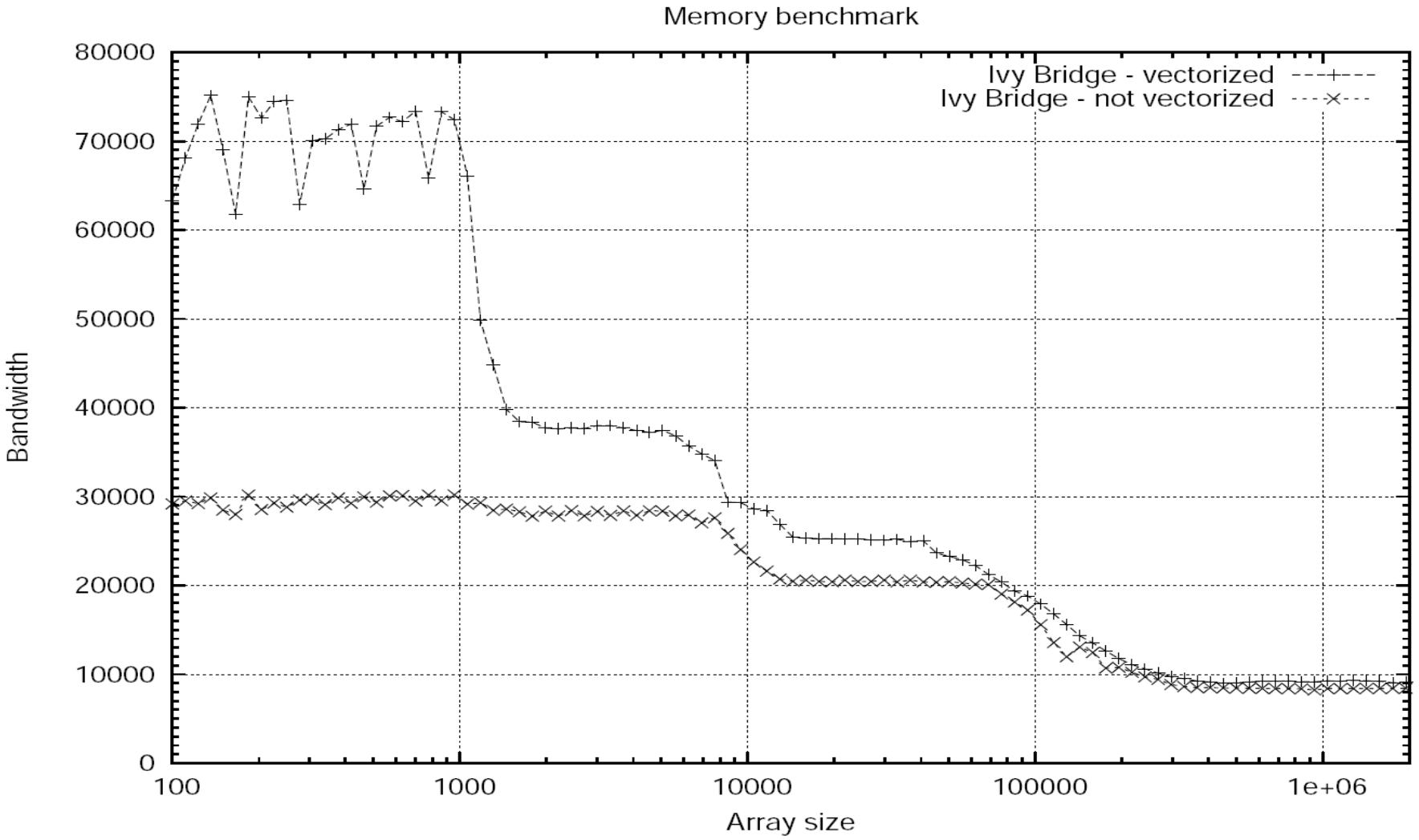




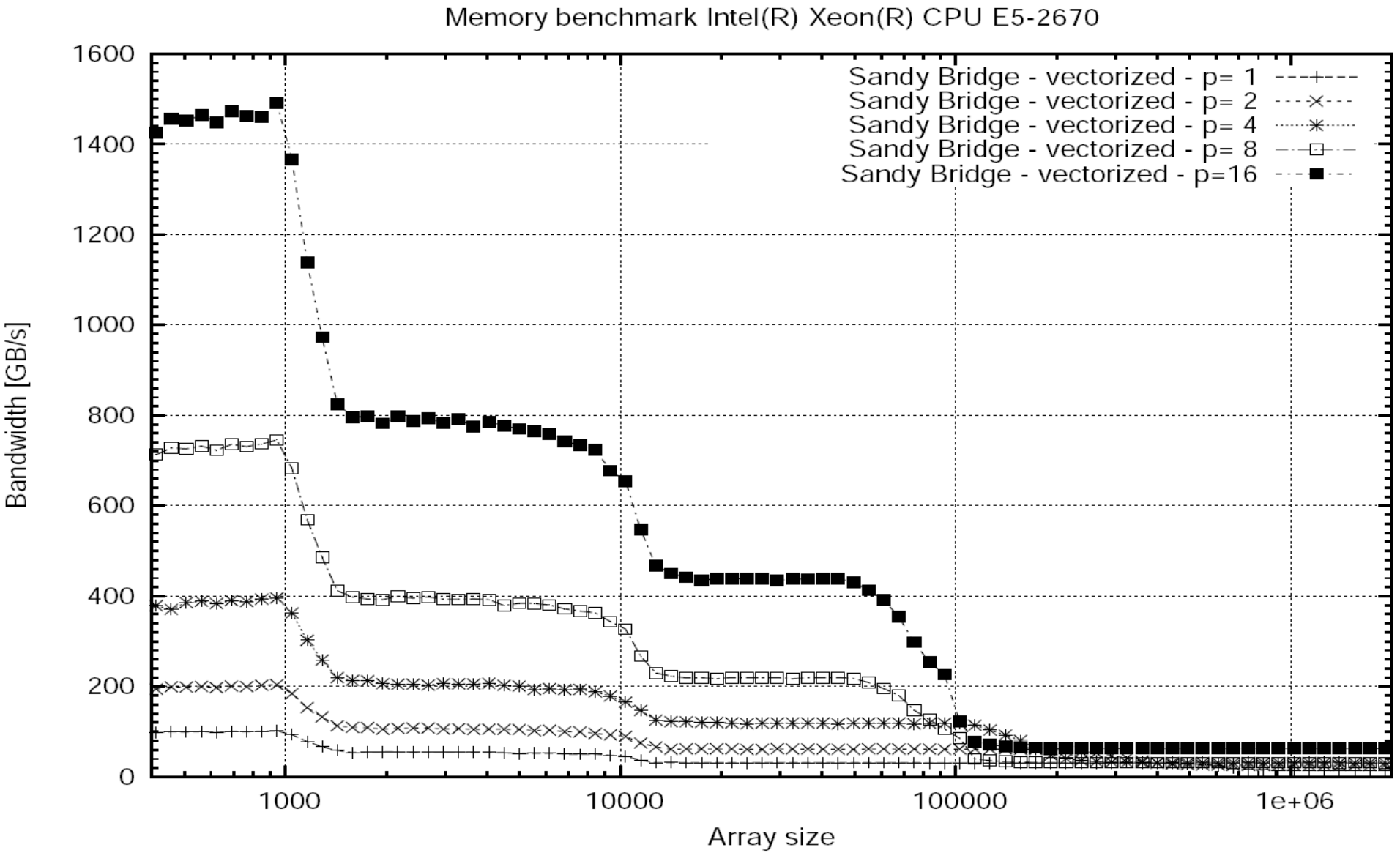
2-socket CPU node



Memory throughput – single thread



Memory throughput - scalability



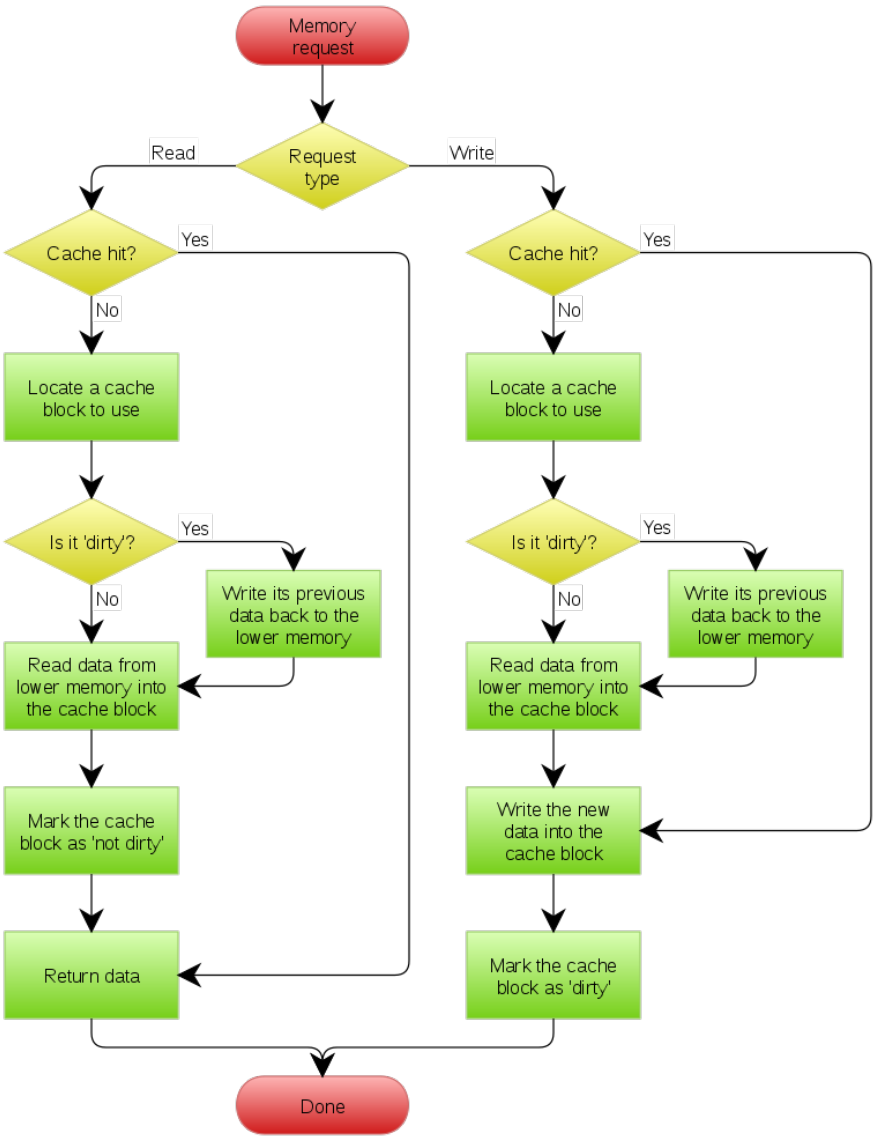
Scalability

- Impediments to intra-node scalability:
  - single thread performance
    - first step in optimization is often single thread optimization
  - system overhead
    - thread management, memory management
    - synchronization for shared data: mutexes, atomic operations, etc.
  - resource exhaustion
    - memory bandwidth, I/O bandwidth
  - thread affinity and resource contention
    - the use of processing power of cores
      - load balancing; SMT, hyperthreading – not really parallel
    - the use of memory hierarchy
      - TLB and cache flushing for context switches
      - NUMA accesses, first touch allocation
      - arrays alignment, cache line contention, false sharing

Cache coherence

- Each value in any cache is a copy (at least initially) of the value in the corresponding main memory location
  - when reading data, caches are just a mean for speeding up the process
  - when writing data (even for single thread execution), there appear different possible strategies:
    - if the memory location being the target of write operation has its copy in a cache:
      - *write-through* – the write operation may update the value in the cache and in the memory
      - *write-back* – the write operation update only the value in the cache and the modification of the value in the memory is postponed, e.g. until the cache line is evicted from the cache
    - if the memory location has no copy in cache:
      - *write-allocate* – first read data into cache then modify it in cache
      - *no-write-allocate* – write data directly to memory

Cache coherence

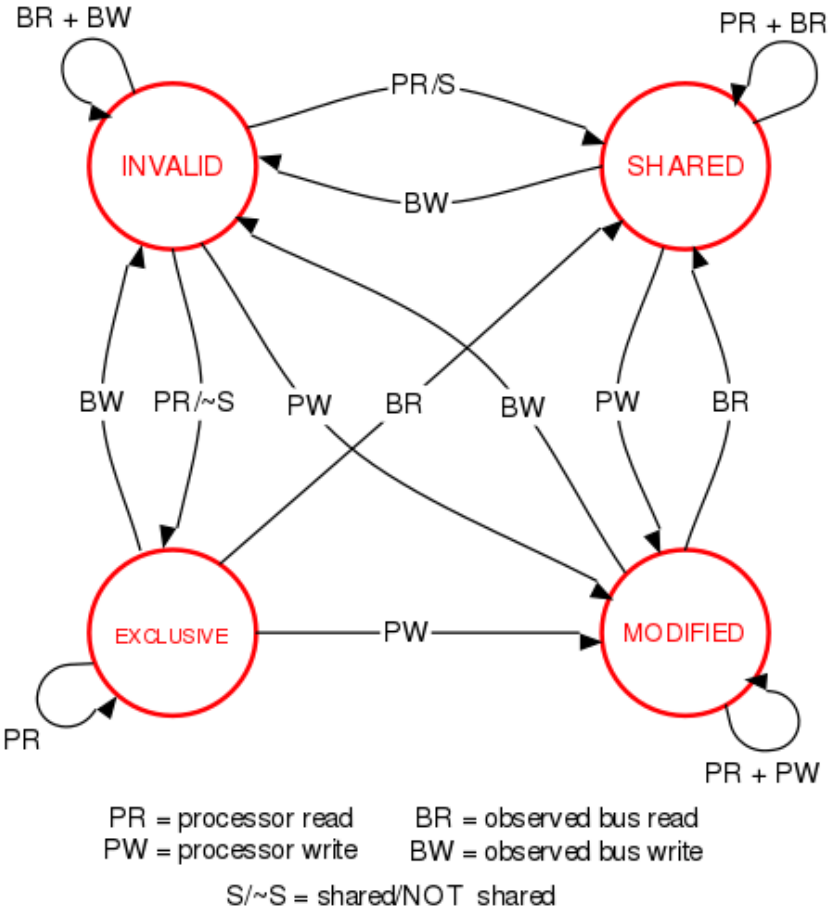


- The most popular writing strategy is the combination of *write-back* and *write-allocate* approaches
  - for this strategy the values corresponding to the same memory location in a cache and in memory can become different
  - there must exist a method for specifying which one is the proper current value
  - the problem is even more important when several caches exist that contain the copies of the same memory location, as is often the case for multi-core multithreading

Cache coherence

- Cache coherence protocol is an algorithm for maintaining the coherent state of caches
  - cache coherence protocol can be based on:
    - snooping (observing) the state of bus by each cache controller
    - putting shared cached data in a separate directory
  - snooping protocols often use the strategy of modifying the state for each cache line, depending on the processor activity and observed bus activity (write invalidate strategy)
    - one of such protocols is MESI protocol with the states: Exclusive, Modified, Shared and Invalid

State diagram of the MESI protocol



Cache coherence

- Cache coherence may lead to unnecessary performance degradation when several threads modify repeatedly different variables (apparently with no dependence) that reside in a memory block corresponding to a single cache line – so called false sharing
  - when one thread modifies its variable it makes the whole cache line invalid for all other threads
  - when another thread wants to modify its own (different) variable it finds the cache line invalid and have to read it again, modify the variable and make the whole cache line invalid for all other threads
  - then the first thread wants to modify its own unknown and the whole process repeats, and so on
  - the described situation leads to many subsequent *reads-for-ownership*, when a cache line is read in order to be modified as owned