

High Performance Computing

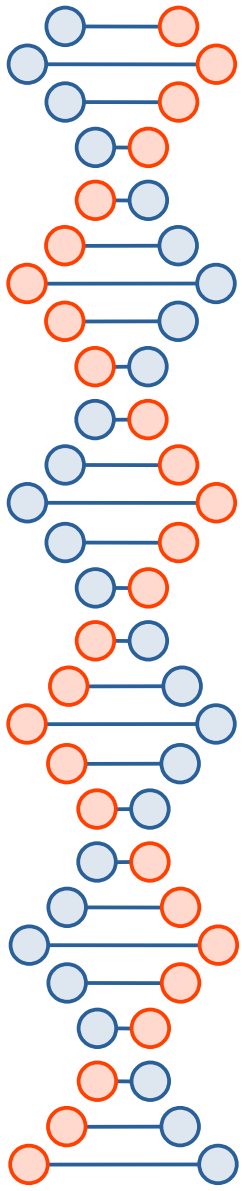
Ch. 4 Parallel Computers

Computer Eng., KMITL
Assoc. Prof. Dr. Surin. K.

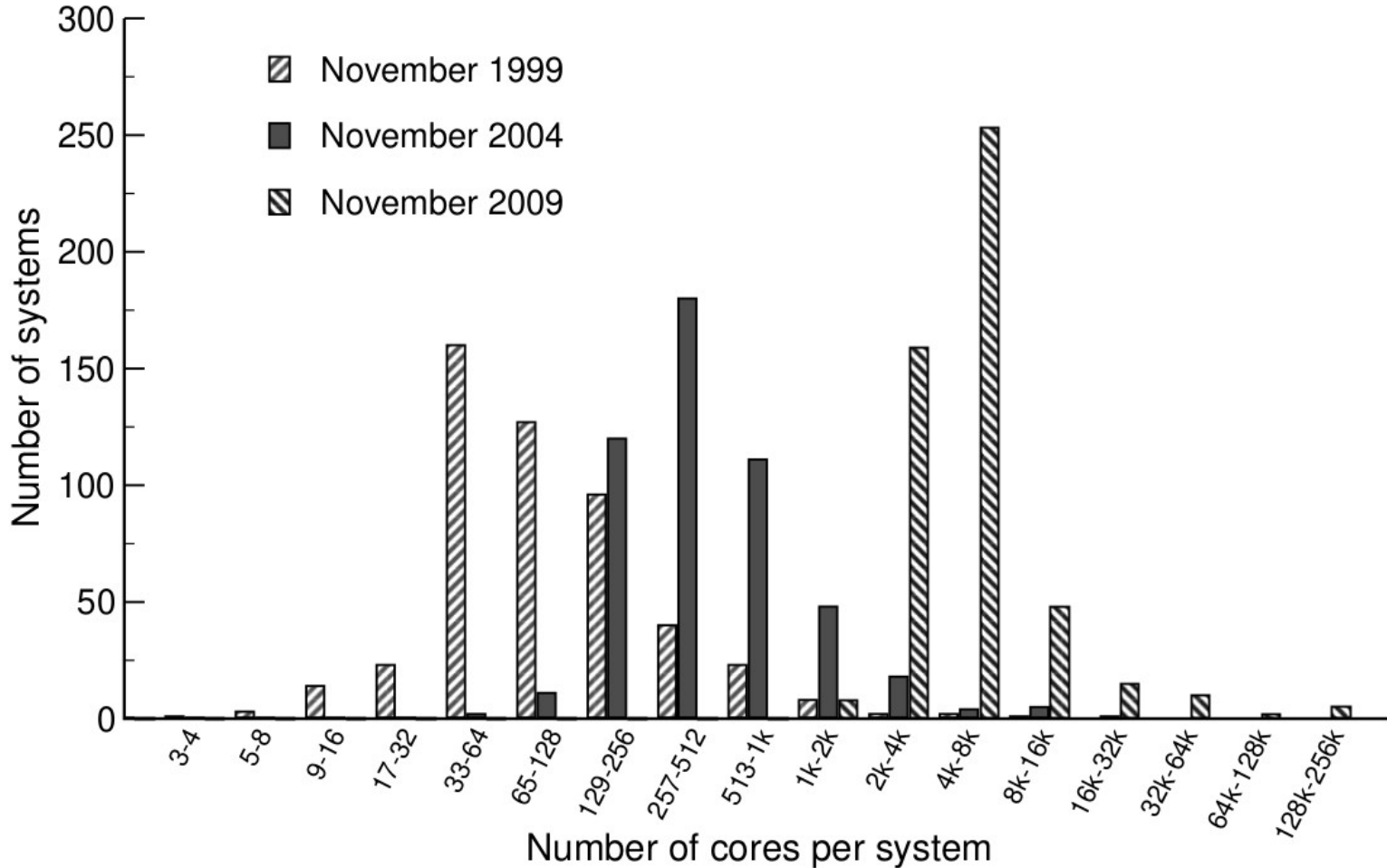


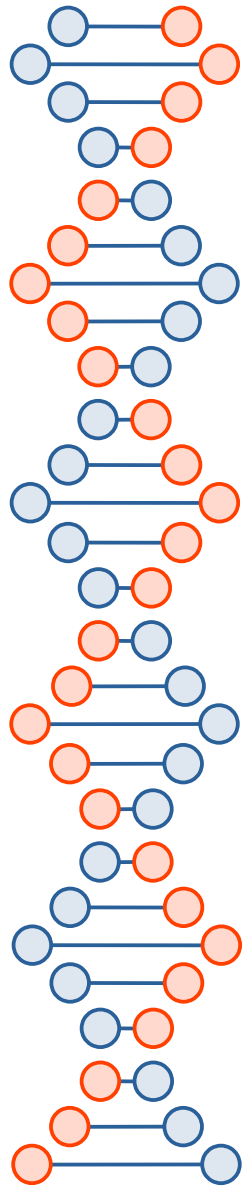
Ch.4 Parallel Computers

- 4.1 Taxonomy of parallel computing paradigms
- 4.2 Shared-memory computers
- 4.3 Distributed-memory computers
- 4.4 Hierarchical (hybrid) systems
- 4.5 Networks



Number of systems versus core count





4.1 Taxonomy of parallel computing paradigms

SIMD *Single Instruction, Multiple Data.* A single instruction stream, either on a single processor (core) or on multiple compute elements, provides parallelism by operating on multiple data streams concurrently. Examples are vector processors (see Section 1.6), the SIMD capabilities of modern superscalar microprocessors (see Section 2.3.3), and Graphics Processing Units (GPUs). Historically, the all but extinct large-scale multiprocessor SIMD parallelism was implemented in Thinking Machines' *Connection Machine* supercomputer [R36].

MIMD *Multiple Instruction, Multiple Data.* Multiple instruction streams on multiple processors (cores) operate on different data items concurrently. The shared-memory and distributed-memory parallel computers described in this chapter are typical examples for the MIMD paradigm.

4.2 Shared-Memory Computers

- *Uniform Memory Access* (UMA) systems exhibit a “flat” memory model: Latency and bandwidth are the same for all processors and all memory locations. This is also called *symmetric multiprocessing* (SMP). At the time of writing, single multicore processor chips (see Section 1.4) are “UMA machines.” However, “cluster on a chip” designs that assign separate memory controllers to different groups of cores on a die are already beginning to appear.
- On *cache-coherent Nonuniform Memory Access* (ccNUMA) machines, memory is *physically distributed* but *logically shared*. The physical layout of such systems is quite similar to the distributed-memory case (see Section 4.3), but network logic makes the aggregated memory of the whole system appear as one single address space. Due to the distributed nature, memory access performance varies depending on which CPU accesses which parts of memory (“local” vs. “remote” access).

Uniform Memory Access (UMA)

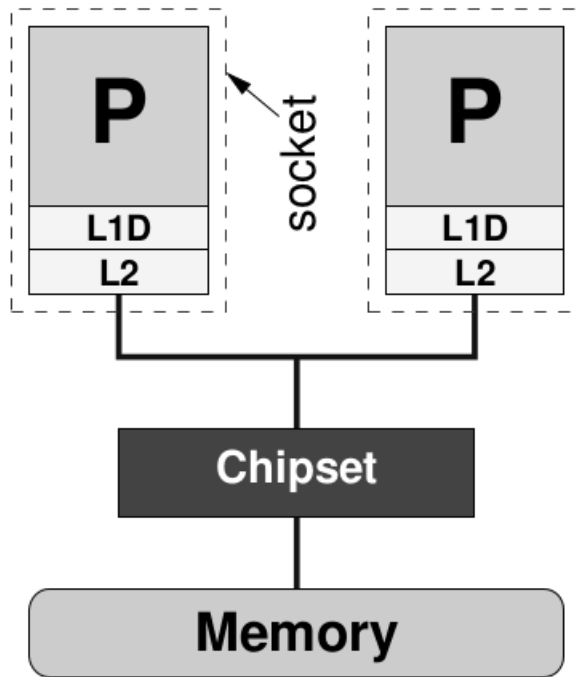


Figure 4.3: A UMA system with two single-core CPUs that share a common frontside bus (FSB).

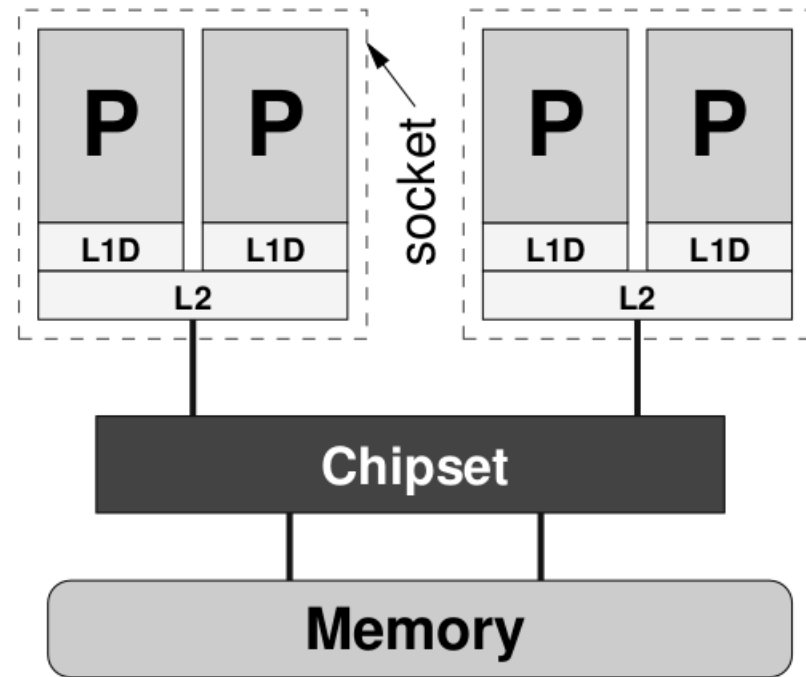
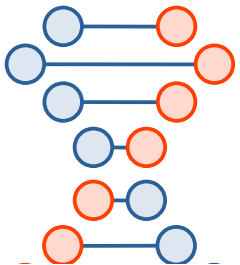
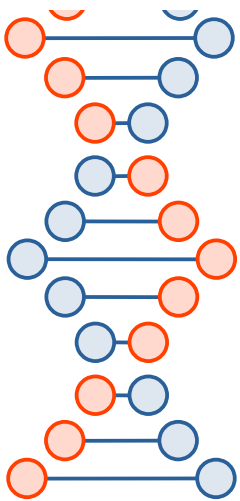


Figure 4.4: A UMA system in which the FSBs of two dual-core chips are connected separately to the chipset.

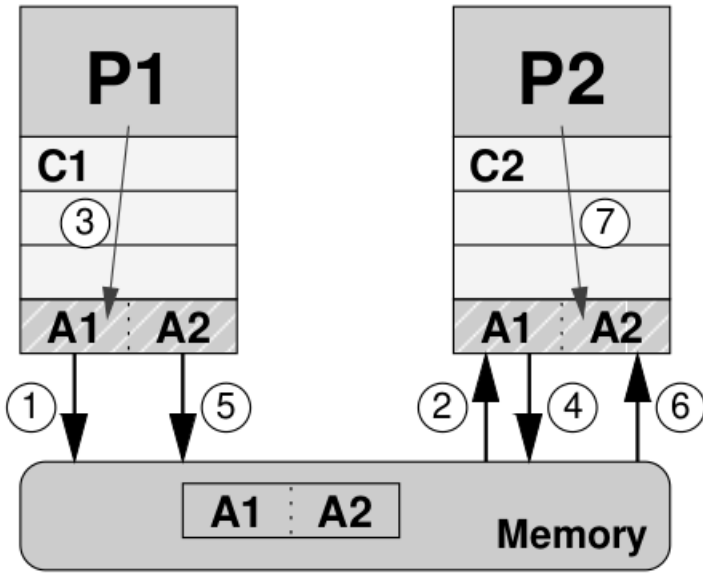


4.2.1 Cache Coherence

Cache coherence mechanisms are required in all cache-based multiprocessor systems, whether they are of the UMA or the ccNUMA kind. This is because copies of the same cache line could potentially reside in several CPU caches. If, e.g., one of those gets modified and evicted to memory, the other caches' contents reflect outdated data. Cache coherence protocols ensure a consistent view of memory under all circumstances.



Cache Coherence Problem



1. C1 requests exclusive CL ownership
2. set CL in C2 to state I
3. CL has state E in C1 → modify A1 in C1 and set to state M
4. C2 requests exclusive CL ownership
5. evict CL from C1 and set to state I
6. load CL to C2 and set to state E
7. modify A2 in C2 and set to state M in C2

Figure 4.2: Two processors P1, P2 modify the two parts A1, A2 of the same cache line in caches C1 and C2. The MESI coherence protocol ensures consistency between cache and memory.



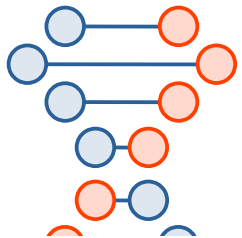
MESI Coherence Protocol

M *modified*: The cache line has been modified in this cache, and it resides in no other cache than this one. Only upon eviction will memory reflect the most current state.

E *exclusive*: The cache line has been read from memory but not (yet) modified. However, it resides in no other cache.

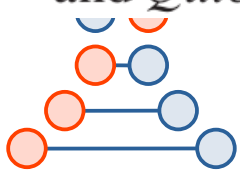
S *shared*: The cache line has been read from memory but not (yet) modified. There may be other copies in other caches of the machine.

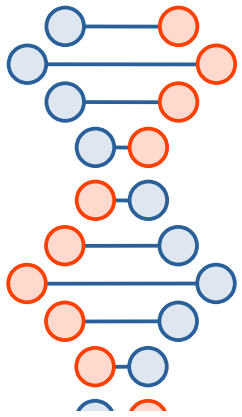
I *invalid*: The cache line does not reflect any sensible data. Under normal circumstances this happens if the cache line was in the shared state and another processor has requested exclusive ownership.



4.2.2 ccNUMA system

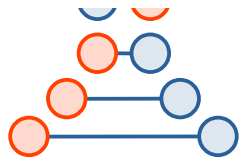
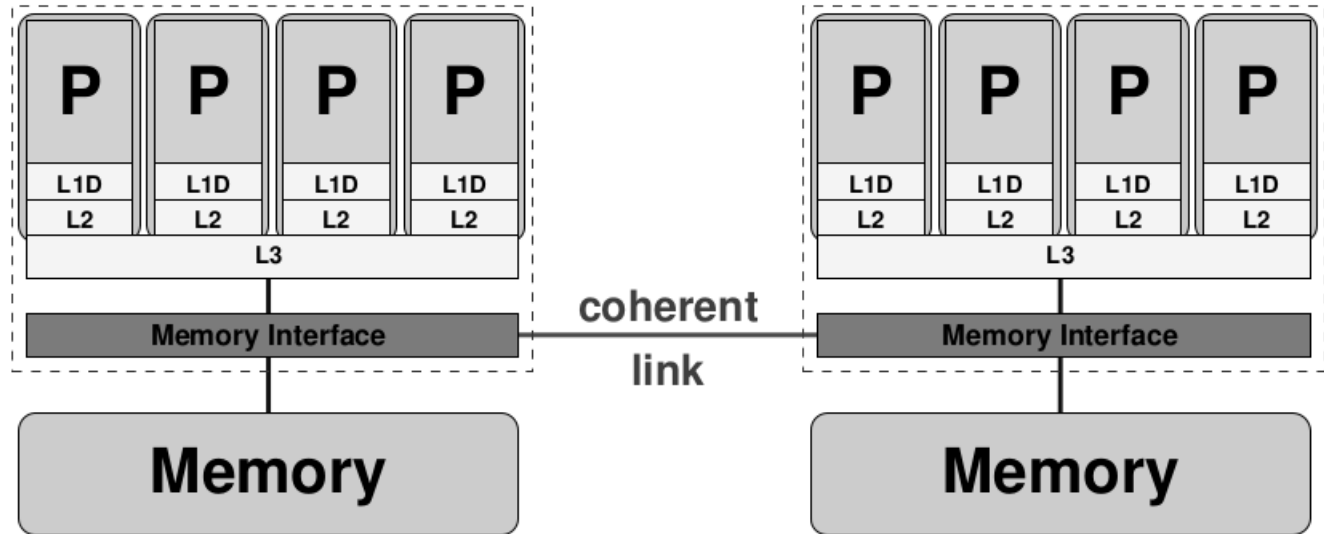
- In ccNUMA, a *locality domain* (LD) is a set of processor cores together with locally connected memory. This memory can be accessed in the most efficient way, i.e., without resorting to a network of any kind. Multiple LDs are linked via a *coherent* interconnect, which allows transparent access from any processor to any other processor's memory. In this sense, a locality domain can be seen as a UMA “building block.” The whole system is still of the shared-memory kind, and runs a single OS instance. Although the ccNUMA principle provides scalable bandwidth for very
- large processor counts, it is also found in inexpensive small two- or four-socket nodes frequently used for HPC clustering (see Figure 4.5). In this particular example two locality domains, i.e., quad-core chips with separate caches and a common interface
- to local memory, are linked using a high-speed connection. *HyperTransport* (HT) and *QuickPath* (QPI) are the current technologies favored by AMD and Intel, respec-





ccNUMA system with Coherent Link

Figure 4.5: A ccNUMA system with two locality domains (one per socket) and eight cores.



ccNUMA system with NUMALink and Router

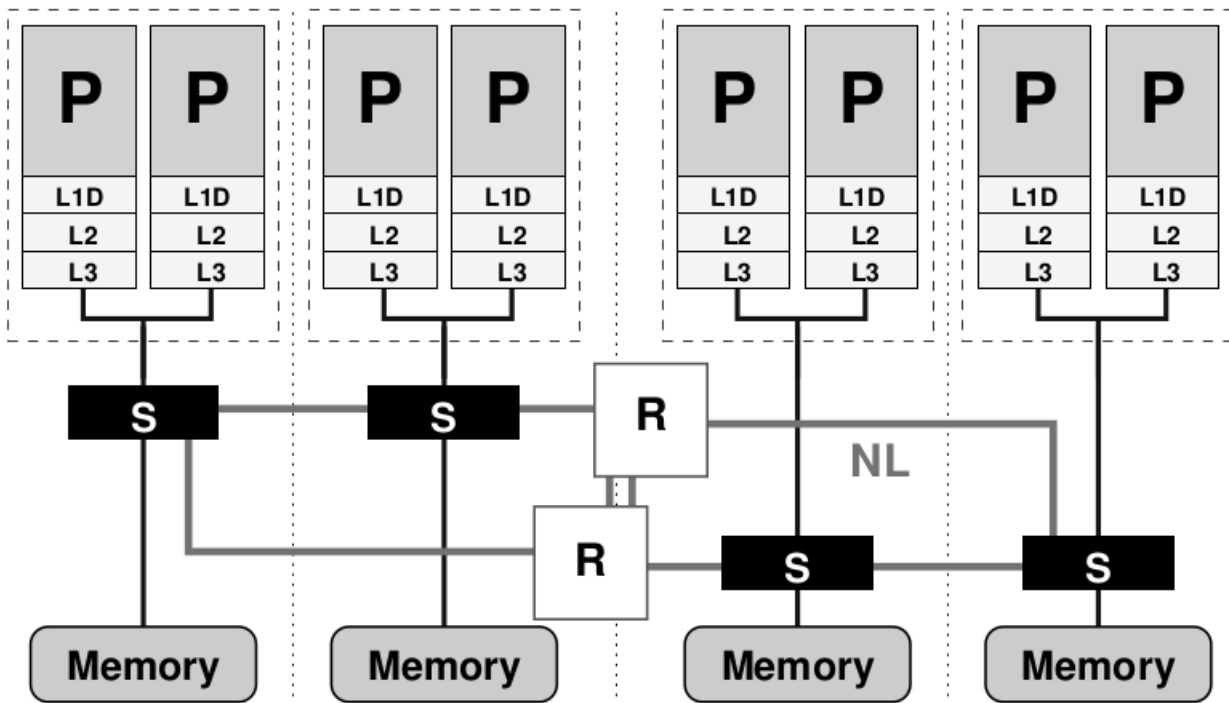
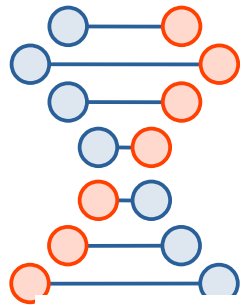
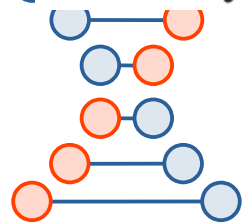


Figure 4.6: A ccNUMA system (SGI Altix) with four locality domains, each comprising one socket with two cores. The LDs are connected via a routed NUMALink (NL) network using routers (R).



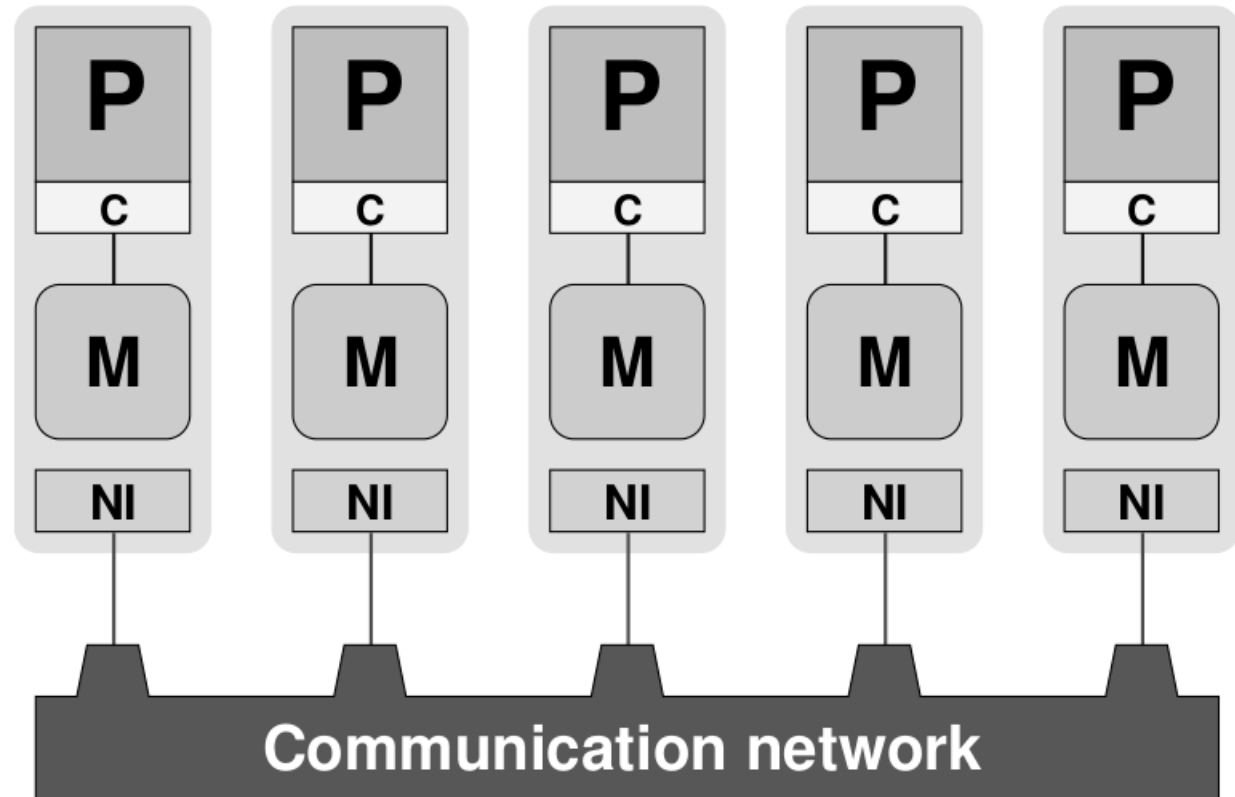
4.3 Distributed Memory Computers

Figure 4.7 shows a simplified block diagram of a distributed-memory parallel computer. Each processor P is connected to exclusive local memory, i.e., no other CPU has direct access to it. Nowadays there are actually no distributed-memory systems any more that implement such a layout. In this respect, the sketch is to be seen as a *programming model* only. For price/performance reasons all parallel machines today, first and foremost the popular PC clusters, consist of a number of shared-memory “compute nodes” with two or more CPUs (see the next section); the “distributed-memory programmer’s” view does not reflect that. It is even possible (and quite common) to use distributed-memory programming on pure shared-memory machines.

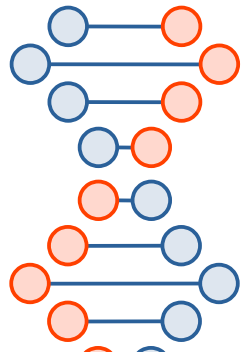


4.3 Distributed Memory Computers

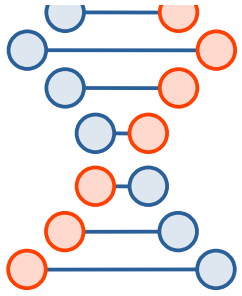
Figure 4.7: Simplified programmer's view, or "programming model," of a distributed-memory parallel computer: Separate processes run on processors (P), communicating via interfaces (NI) over some network. No process can access another process' memory (M) directly, although processors may reside in shared memory.



4.4 Hybrid Systems



Parallel computers with hierarchical structures as described above are also called *hybrids*. The concept is actually more generic and can also be used to categorize any system with a mixture of available programming paradigms on different hardware layers. Prominent examples are clusters built from nodes that contain, besides the “usual” multicore processors, additional *accelerator hardware*, ranging from application-specific add-on cards to GPUs (graphics processing units), FPGAs (field-programmable gate arrays), ASICs (application specific integrated circuits), co-processors, etc.



4.4 Hybrid Systems

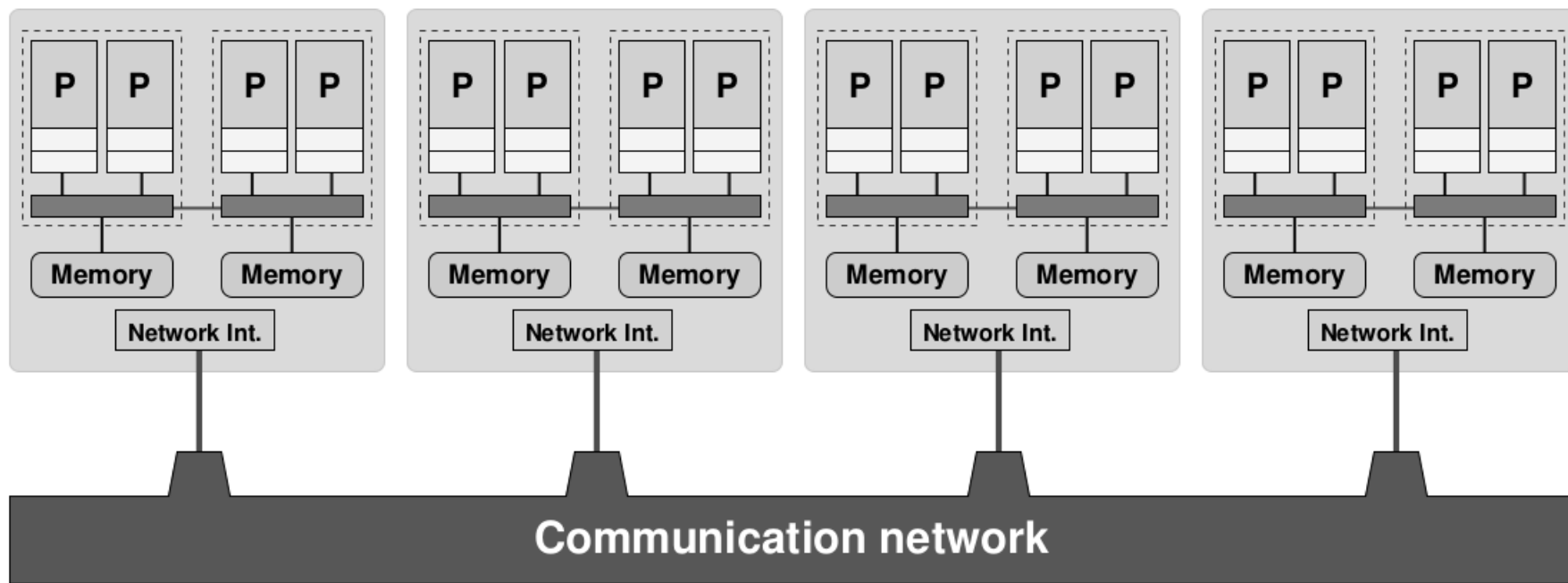
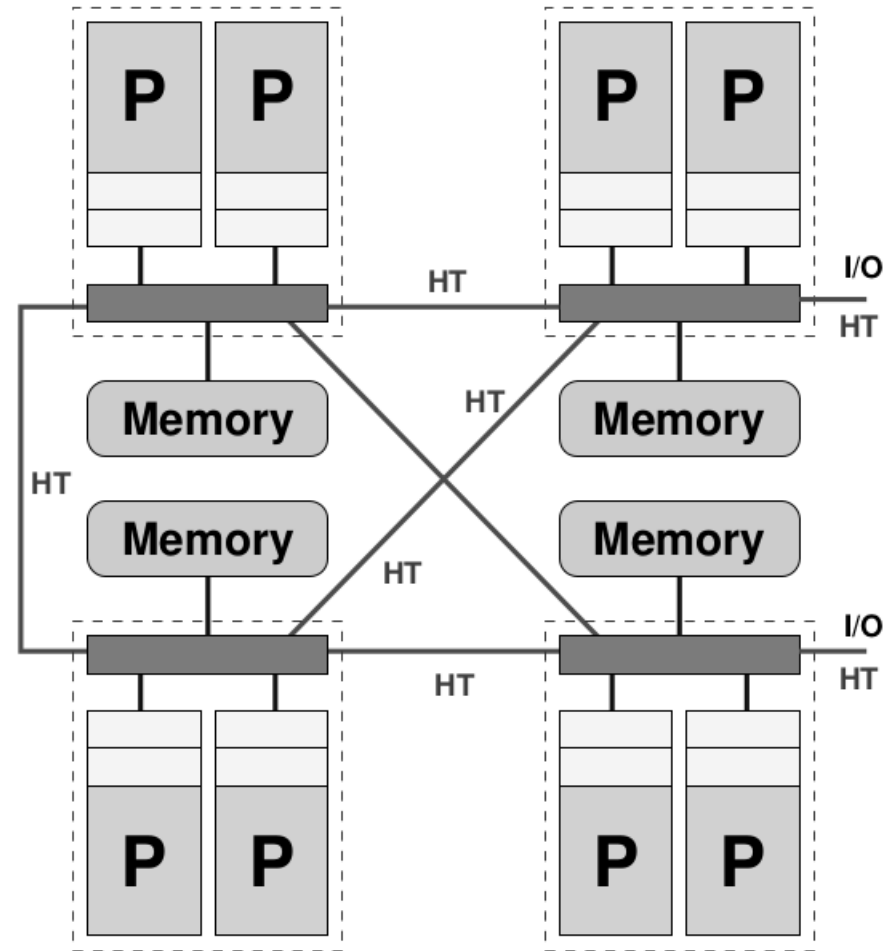
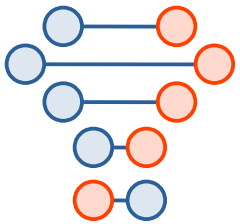


Figure 4.8: Typical hybrid system with shared-memory nodes (ccNUMA type). Two-socket building blocks represent the price vs. performance “sweet spot” and are thus found in many commodity clusters.

4.4 Hybrid Systems

Figure 4.19: A four-socket ccNUMA system with a HyperTransport-based mesh network. Each socket has only three HT links, so the network has to be heterogeneous in order to accommodate I/O connections and still utilize all provided HT ports.





4.5 Networks

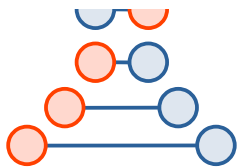
Point-to-point connections

Whatever the underlying hardware may be, the communication characteristics of a single point-to-point connection can usually be described by a simple model: Assuming that the total transfer time for a message of size N [bytes] is composed of latency and streaming parts,

$$T = T_\ell + \frac{N}{B} \quad (4.1)$$

and B being the maximum (asymptotic) network bandwidth in MBytes/sec, the effective bandwidth is

$$B_{\text{eff}} = \frac{N}{T_\ell + \frac{N}{B}} \quad (4.2)$$



4.5 Networks

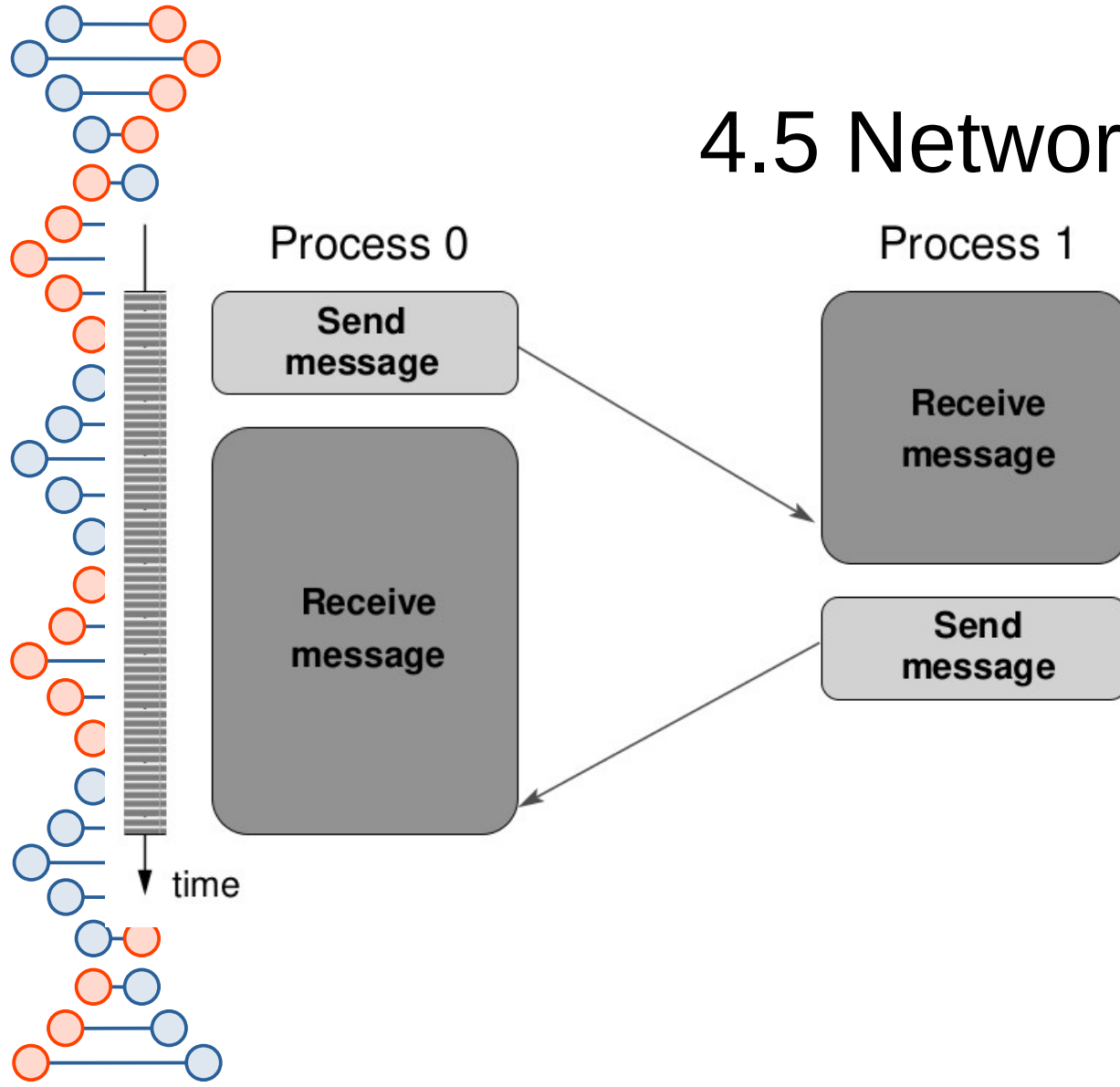


Figure 4.9: Timeline for a “Ping-Pong” data exchange between two processes. PingPong reports the time it takes for a message of length N bytes to travel from process 0 to process 1 and back.



4.5 Networks

```
1 myID = get_process_ID()
2 if(myID.eq.0) then
3   targetID = 1
4   S = get_walltime()
5   call Send_message(buffer,N,targetID)
6   call Receive_message(buffer,N,targetID)
7   E = get_walltime()
8   MBYTES = 2*N/(E-S)/1.d6      ! MBytes/sec rate
9   TIME    = (E-S)/2*1.d6      ! transfer time in microseconds
10                                     ! for single message
11 else
12   targetID = 0
13   call Receive_message(buffer,N,targetID)
14   call Send_message(buffer,N,targetID)
15 endif
```



4.5 Networks

In contrast to bandwidth limitations, which are usually set by the physical parameters of data links, latency is often composed of several contributions:

- All data transmission protocols have some overhead in the form of administrative data like message headers, etc.
- Some protocols (like, e.g., TCP/IP as used over Ethernet) define minimum message sizes, so even if the application sends a single byte, a small “frame” of $N > 1$ bytes is transmitted.
- Initiating a message transfer is a complicated process that involves multiple software layers, depending on the complexity of the protocol. Each software layer adds to latency.
- Standard PC hardware as frequently used in clusters is not optimized towards low-latency I/O.

4.5 Networks

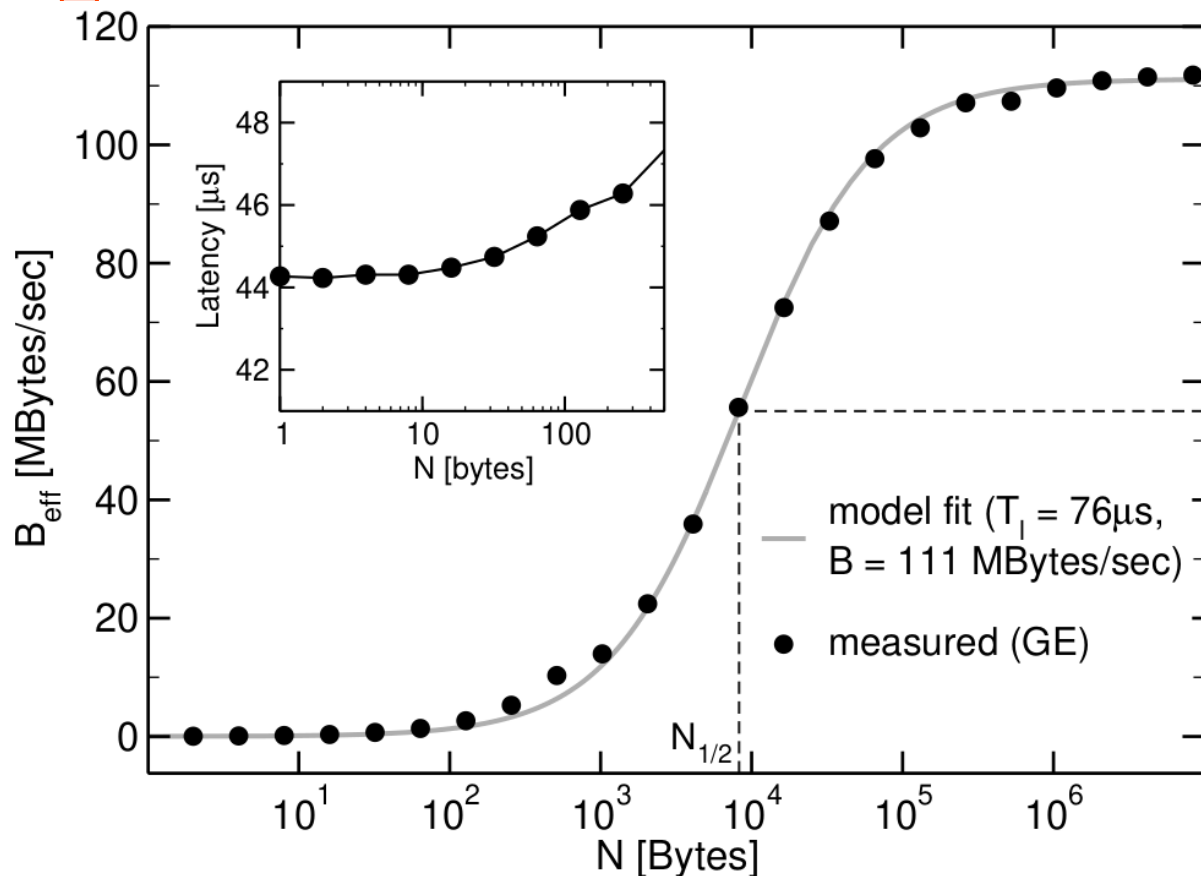


Figure 4.10: Fit of the model for effective bandwidth (4.2) to data measured on a GigE network. The fit cannot accurately reproduce the measured value of T_ℓ (see text). $N_{1/2}$ is the message length at which half of the saturation bandwidth is reached (dashed line).

4.5 Networks

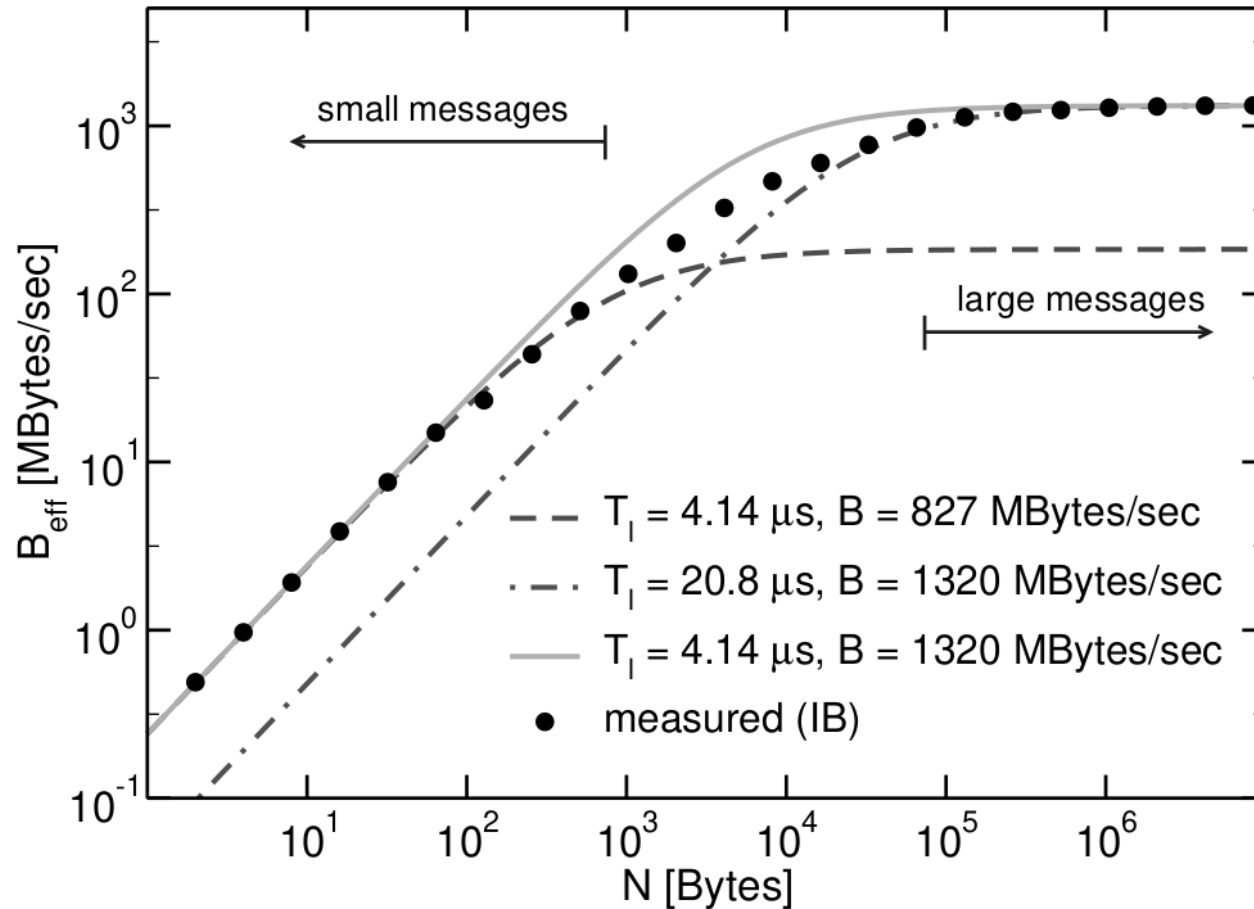
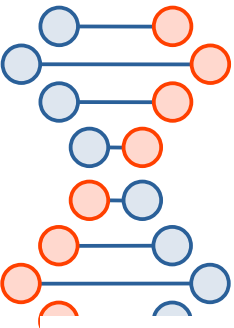
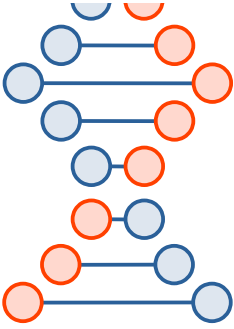


Figure 4.11: Fits of the model for effective bandwidth (4.2) to data measured on a DDR InfiniBand network. “Good” fits for asymptotic bandwidth (dotted-dashed) and latency (dashed) are shown separately, together with a fit function that unifies both (solid).

4.5 Networks



on either end of the scale. Figure 4.11 shows measured PingPong data for a *DDR-InfiniBand* network. Both axes have been scaled logarithmically in this case because this makes it easier to judge the fit quality on all scales. The dotted-dashed and dashed curves have been obtained by restricting the fit to the large- and small-message-size regimes, respectively. The former thus yields a good estimate for B , while the latter allows quite precise determination of T_ℓ . Using the fit function (4.2) with those two parameters combined (solid curve) reveals that the model produces mediocre results for intermediate message sizes. There can be many reasons for such a failure; com-



4.5.2 Buses

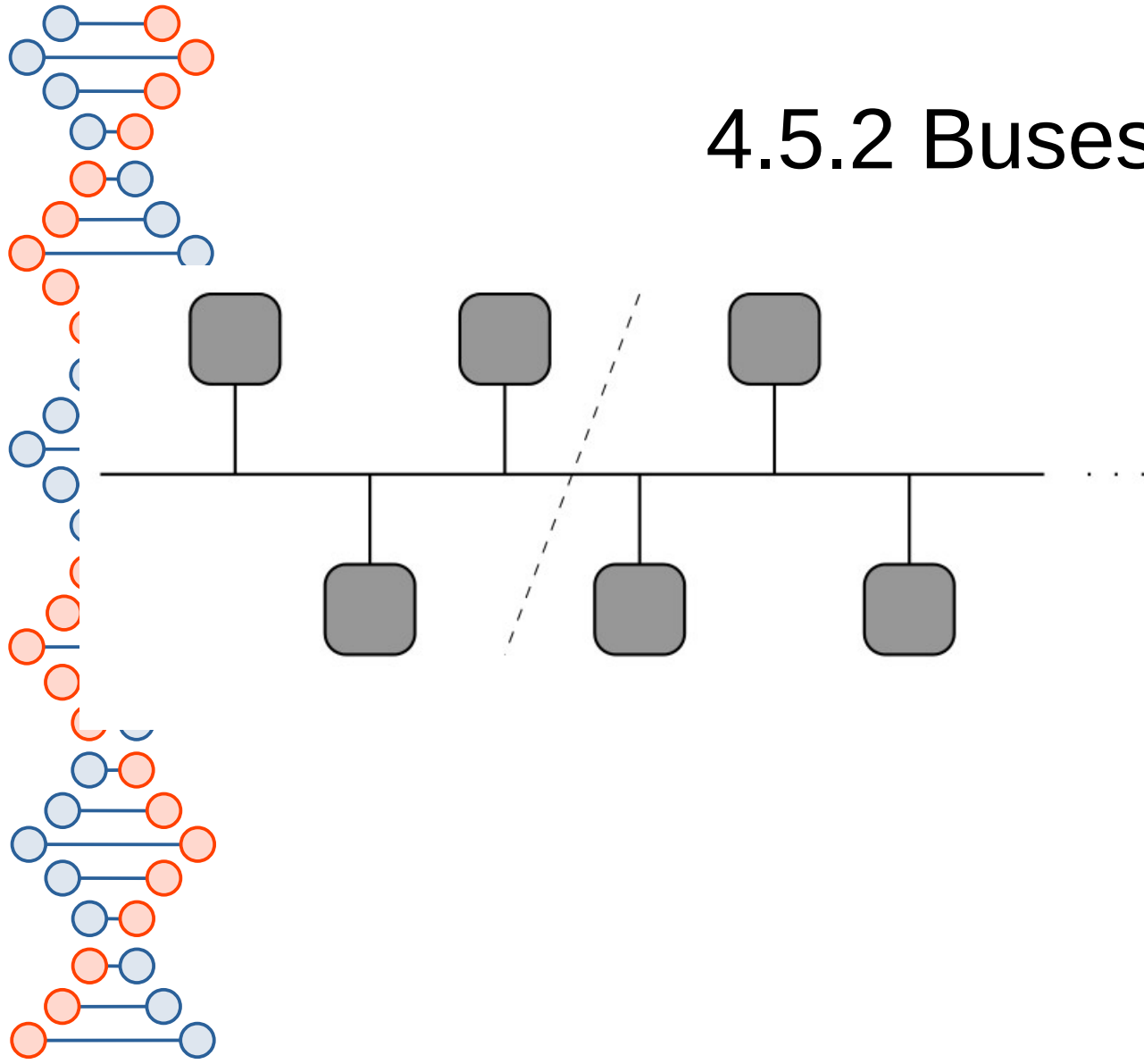
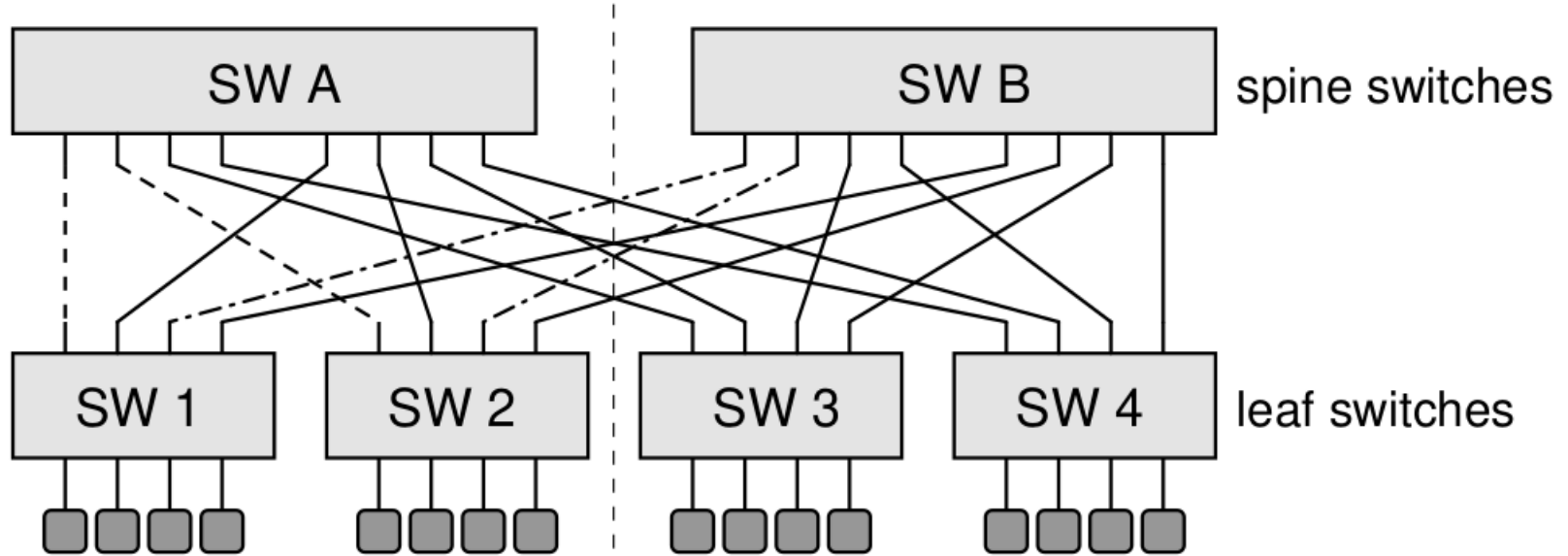
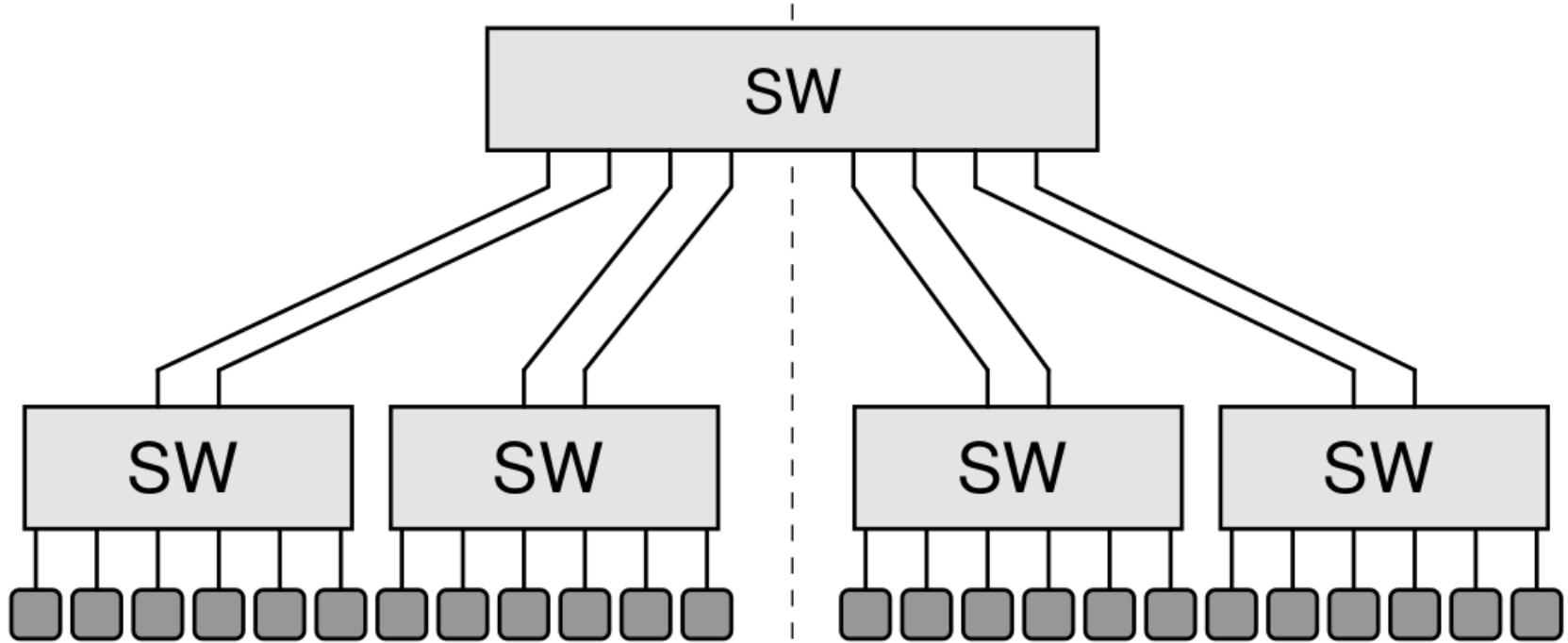


Figure 4.13: A bus network (shared medium). Only one device can use the bus at any time, and bisection bandwidth is independent of the number of nodes.

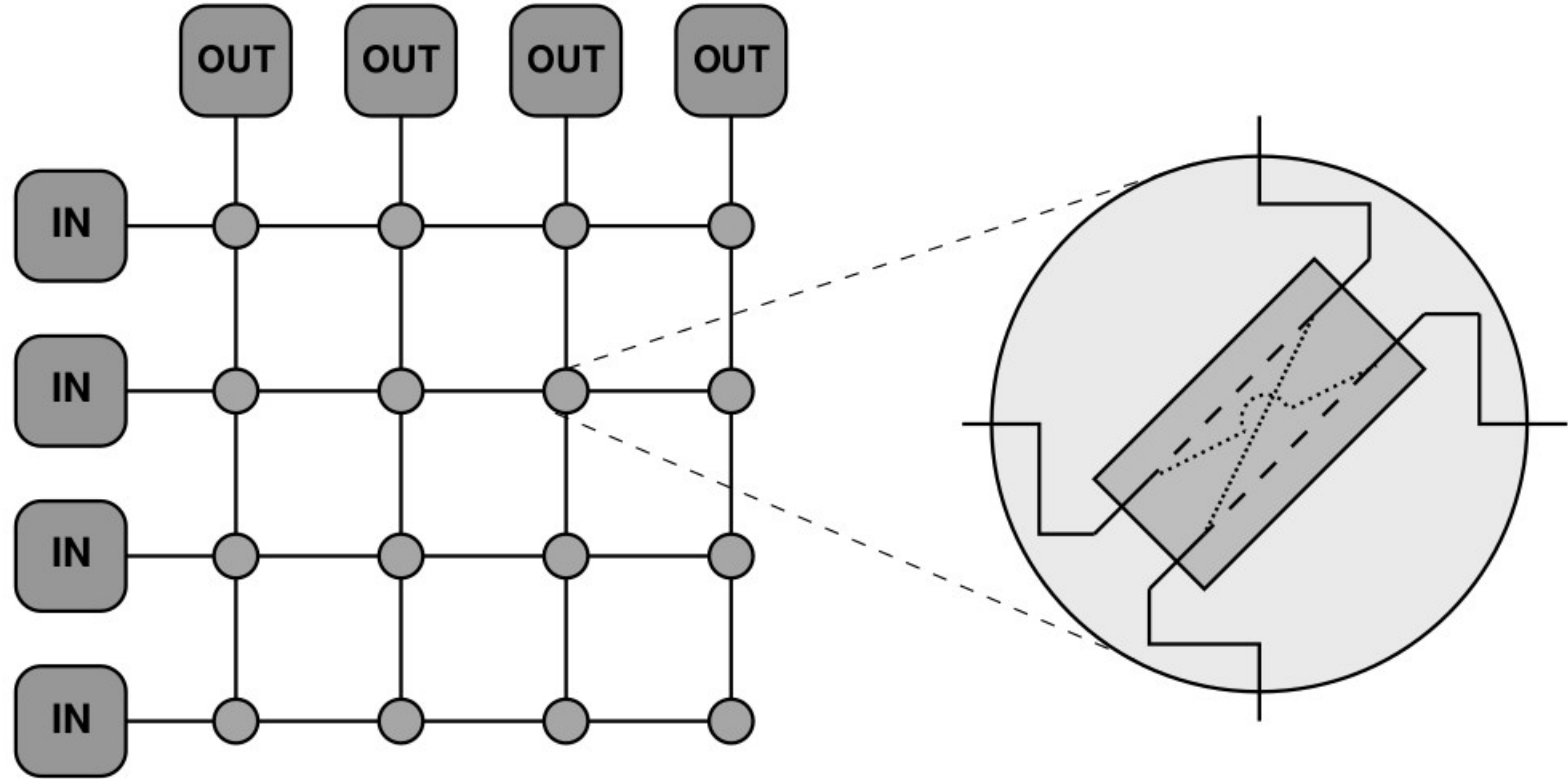
4.5.3 Switched and fat-tree networks

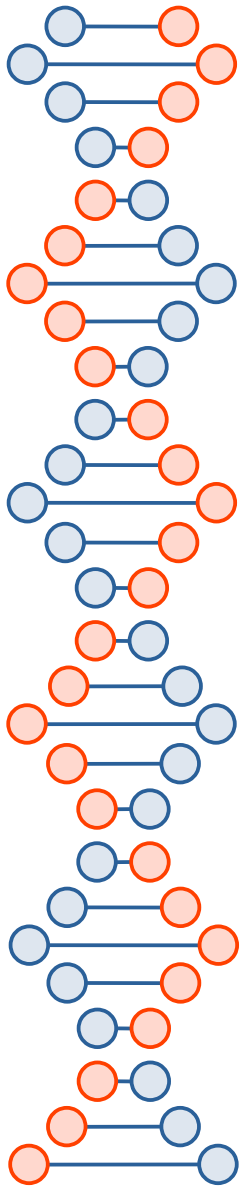


4.5.3 Switched and fat-tree networks

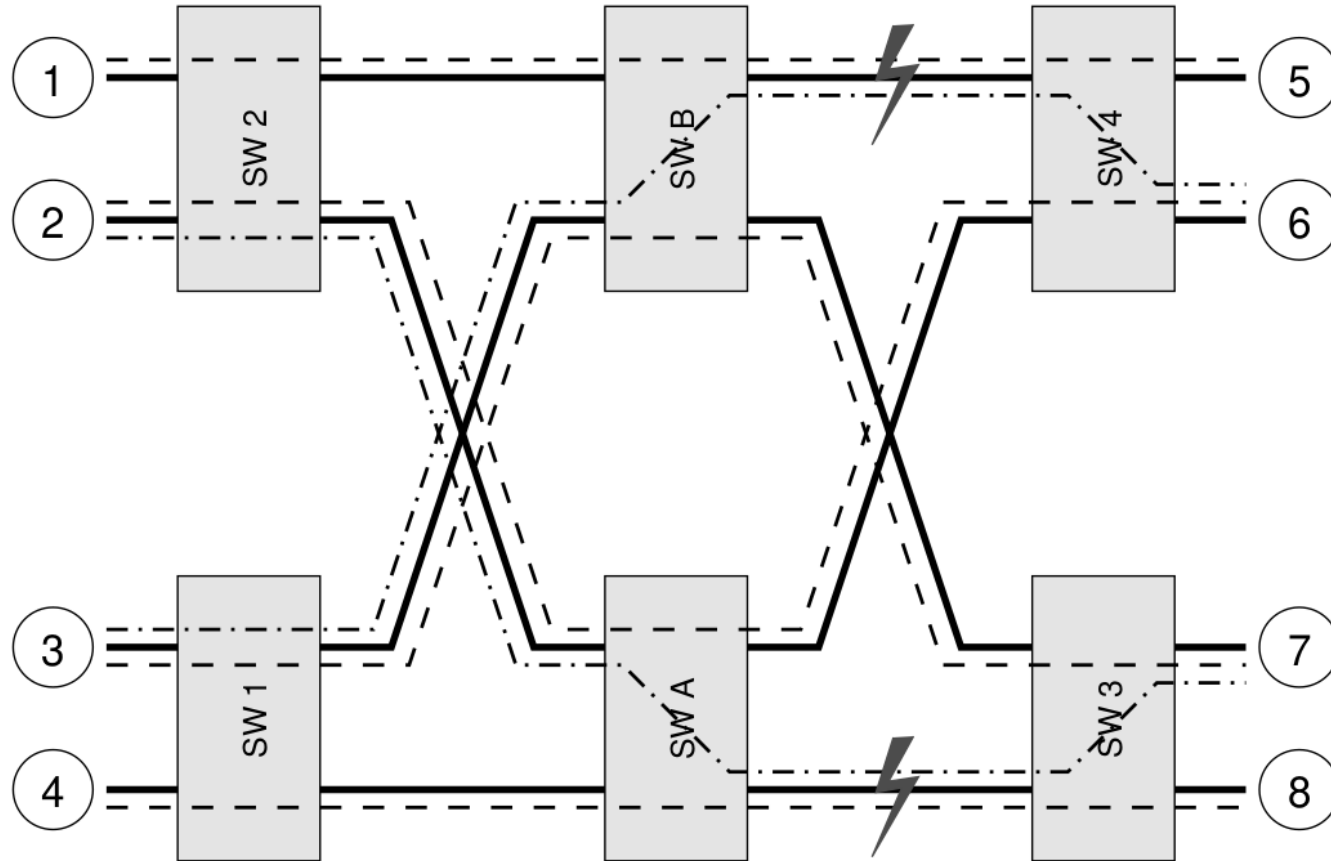


4.5.3 Switched and fat-tree networks





4.5.3 Switched and fat-tree networks



4.5.4 Mesh Networks

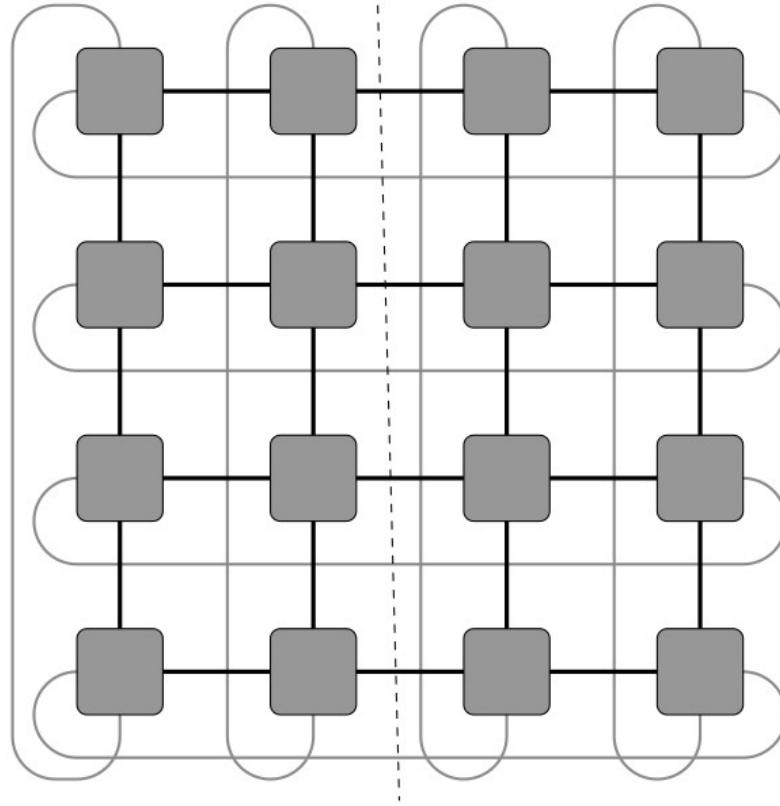


Figure 4.18: A two-dimensional (square) torus network. Bisection bandwidth scales like \sqrt{N} in this case.