Chapter 4

Parallel computers

We speak of *parallel computing* whenever a number of "compute elements" (cores) solve a problem in a cooperative way. All modern supercomputer architectures depend heavily on parallelism, and the number of CPUs in large-scale supercomputers increases steadily. A common measure for supercomputer "speed" has been established by the Top500 list [W121], which is published twice a year and ranks parallel computers based on their performance in the LINPACK benchmark. LINPACK solves a dense system of linear equations of unspecified size. It is not generally accepted as a good metric because it covers only a single architectural aspect (peak performance). Although other, more realistic alternatives like the HPC Challenge benchmarks [W122] have been proposed, the simplicity of LINPACK and its ease of use through efficient open-source implementations have preserved its dominance in the Top500 ranking for nearly two decades now. Nevertheless, the list can still serve

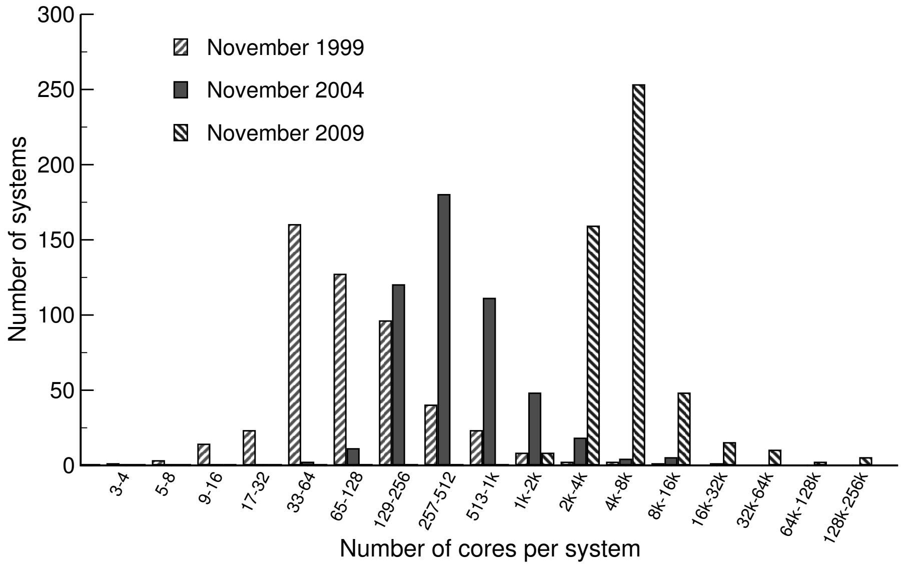


Figure 4.1: Number of systems versus core count in the November 1999, 2004, and 2009 Top500 lists. The average number of CPUs has grown 50-fold in ten years. Between 2004 and 2009, the advent of multicore chips resulted in a dramatic boost in typical core counts. Data taken from [W121].

as an important indicator for trends in supercomputing. The main tendency is clearly visible from a comparison of processor number distributions in Top500 systems (see Figure 4.1): Top of the line HPC systems do not rely on Moore's Law alone for performance but *parallelism* becomes more important every year. This trend has been accelerating recently by the advent of multicore processors — apart from the occasional parallel vector computer, the latest lists contain no single-core systems any more (see also Section 1.4). We can certainly provide no complete overview on current parallel computer technology, but recommend the regularly updated *Overview of recent supercomputers* by van der Steen and Dongarra [W123].

In this chapter we will give an introduction to the fundamental variants of parallel computers: the shared-memory and the distributed-memory type. Both utilize networks for communication between processors or, more generally, "computing elements," so we will outline the basic design rules and performance characteristics for the common types of networks as well.

4.1 Taxonomy of parallel computing paradigms

A widely used taxonomy for describing the amount of concurrent control and data streams present in a parallel architecture was proposed by Flynn [R38]. The dominating concepts today are the SIMD and MIMD variants:

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.

There are actually two more categories, called *SISD* (Single Instruction Single Data) and *MISD* (Multiple Instruction Single Data), the former describing conventional, nonparallel, single-processor execution following the original pattern of the stored-program digital computer (see Section 1.1), while the latter is not regarded as a useful paradigm in practice.

Strictly processor-based instruction-level parallelism as employed in superscalar, pipelined execution (see Sections 1.2.3 and 1.2.4) is not included in this categorization, although one may argue that it could count as MIMD. However, in what follows we will restrict ourselves to the multiprocessor MIMD parallelism built into shared- and distributed-memory parallel computers.

4.2 Shared-memory computers

A *shared-memory parallel computer* is a system in which a number of CPUs work on a common, shared physical address space. Although transparent to the programmer as far as functionality is concerned, there are two varieties of shared-memory systems that have very different performance characteristics in terms of main memory access:

- *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.

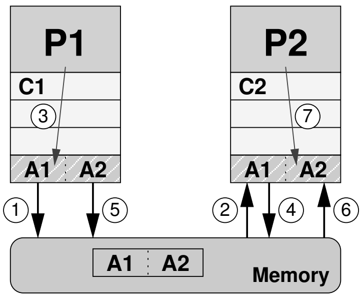
- On *cache-coherent Nonuniform Memory Access* (ccNUMA) machines, memory is *physically distributed* but *logically shared*. The physical layout of such systems is quite similar to the distributed-memory case (see Section 4.3), but network logic makes the aggregated memory of the whole system appear 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).

With multiple CPUs, copies of the same cache line may reside in different caches, probably in modified state. So for both above varieties, *cache coherence protocols* must guarantee consistency between cached data and data in memory at all times. Details about UMA, ccNUMA, and cache coherence mechanisms are provided in the following sections. The dominating shared-memory programming model in scientific computing, OpenMP, will be introduced in Chapter 6.

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.

Figure 4.2 shows an example on two processors P1 and P2 with respective caches C1 and C2. Each cache line holds two items. Two neighboring items A1 and A2 in memory belong to the same cache line and are modified by P1 and P2, respectively. Without cache coherence, each cache would read the line from memory, A1 would get modified in C1, A2 would get modified in C2 and some time later both modified copies of the cache line would have to be evicted. As all memory traffic is handled in



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.

chunks of cache line size, there is no way to determine the correct values of A1 and A2 in memory.

Under control of cache coherence logic this discrepancy can be avoided. As an example we pick the MESI protocol, which draws its name from the four possible states a cache line can assume:

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.

The order of events is depicted in Figure 4.2. The question arises how a cache line in state M is notified when it should be evicted because another cache needs to read the most current data. Similarly, cache lines in state S or E must be invalidated if another cache requests exclusive ownership. In small systems a *bus snoop* is used to achieve this: Whenever notification of other caches seems in order, the originating cache *broadcasts* the corresponding cache line address through the system, and all caches "snoop" the bus and react accordingly. While simple to implement, this method has the crucial drawback that address broadcasts pollute the system buses and reduce available bandwidth for "useful" memory accesses. A separate network for coherence traffic can alleviate this effect but is not always practicable.

A better alternative, usually applied in larger ccNUMA machines, is a *directory-based* protocol where bus logic like chipsets or memory interfaces keep track of the

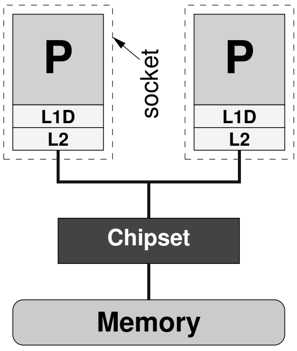


Figure 4.3: A UMA system with two single-core CPUs that share a common frontside bus (FSB).

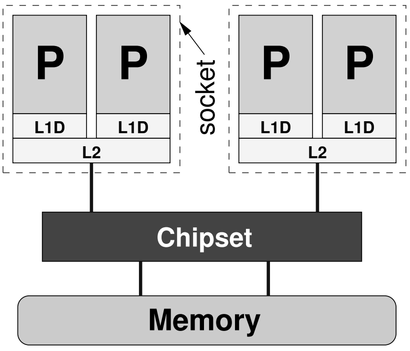


Figure 4.4: A UMA system in which the FSBs of two dual-core chips are connected separately to the chipset.

location and state of each cache line in the system. This uses up some small part of main memory or cache, but the advantage is that state changes of cache lines are transmitted only to those caches that actually require them. This greatly reduces coherence traffic through the system. Today even workstation chipsets implement "snoop filters" that serve the same purpose.

Coherence traffic can severely hurt application performance if the same cache line is modified frequently by different processors (*false sharing*). Section 7.2.4 will give hints for avoiding false sharing in user code.

4.2.2 UMA

The simplest implementation of a UMA system is a dual-core processor, in which two CPUs on one chip share a single path to memory. It is very common in high performance computing to use more than one chip in a compute node, be they single-core or multicore.

In Figure 4.3 two (single-core) processors, each in its own socket, communicate and access memory over a common bus, the so-called *frontside bus* (FSB). All arbitration protocols required to make this work are already built into the CPUs. The chipset (often termed "northbridge") is responsible for driving the memory modules and connects to other parts of the node like I/O subsystems. This kind of design is outdated and is not used any more in modern systems.

In Figure 4.4, two dual-core chips connect to the chipset, each with its own FSB. The chipset plays an important role in enforcing cache coherence and also mediates the connection to memory. In principle, a system like this could be designed so that the bandwidth from chipset to memory matches the aggregated bandwidth of the frontside buses. Each chip features a separate L1 on each core and a dual-core L2 group. The arrangement of cores, caches, and sockets make the system inherently *anisotropic*, i.e., the "distance" between one core and another varies depending on whether they are on the same socket or not. With large many-core processors com-

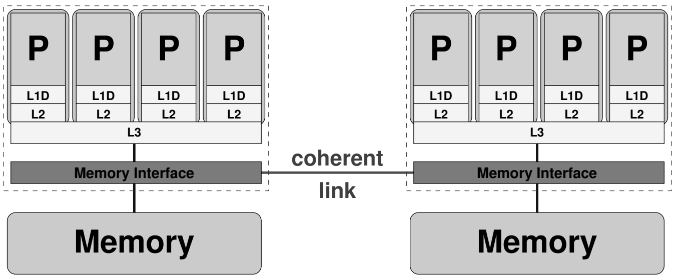


Figure 4.5: A ccNUMA system with two locality domains (one per socket) and eight cores.

prising multilevel cache groups, the situation gets more complex still. See Section 1.4 for more information about shared caches and the consequences of anisotropy.

The general problem of UMA systems is that bandwidth bottlenecks are bound to occur when the number of sockets (or FSBs) is larger than a certain limit. In very simple designs like the one in Figure 4.3, a common *memory bus* is used that can only transfer data to one CPU at a time (this is also the case for all multicore chips available today but may change in the future).

In order to maintain scalability of memory bandwidth with CPU number, non-blocking *crossbar switches* can be built that establish point-to-point connections between sockets and memory modules, similar to the chipset in Figure 4.4. Due to the very large aggregated bandwidths those become very expensive for a larger number of sockets. At the time of writing, the largest UMA systems with scalable bandwidth (the NEC SX-9 vector nodes) have sixteen sockets. This problem can only be solved by giving up the UMA principle.

4.2.3 ccNUMA

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, respectively, but other solutions do exist. Apart from the minor peculiarity that the sockets can drive memory directly, making separate interface chips obsolete, the intersocket link can mediate direct, cache-coherent memory accesses. From the programmer's point of view this mechanism is transparent: All the required protocols are handled by hardware.

Figure 4.6 shows another approach to ccNUMA that is flexible enough to scale

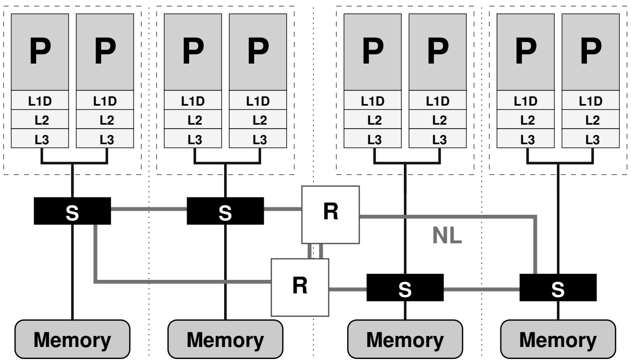


Figure 4.6: A ccNUMA system (SGI Altix) with four locality domains, each comprising one socket with two cores. The LDs are connected via a routed NUMALink (NL) network using routers (R).

to large machines. It is used in Intel-based SGI Altix systems with up to thousands of cores in a single address space and a single OS instance. Each processor socket is connected to a communication interface (S), which provides memory access as well as connectivity to the proprietary *NUMALink* (NL) network. The NL network relies on routers (R) to switch connections for nonlocal access. As with HyperTransport and QuickPath, the NL hardware allows for transparent access to the whole address space of the machine from all cores. Although shown here only with four sockets, multilevel router fabrics can be built that scale up to hundreds of CPUs. It must, however, be noted that each piece of hardware inserted into a data connection (communication interfaces, routers) adds to latency, making access characteristics very inhomogeneous across the system. Furthermore, providing wire-equivalent speed and nonblocking bandwidth for remote memory access in large systems is extremely expensive. For these reasons, large supercomputers and cost-effective smaller clusters are always made from shared-memory building blocks (usually of the ccNUMA type) that are connected via some network without ccNUMA capabilities. See Sections 4.3 and 4.4 for details.

In all ccNUMA designs, network connections must have bandwidth and latency characteristics that are at least the same order of magnitude as for local memory. Although this is the case for all contemporary systems, even a penalty factor of two for nonlocal transfers can badly hurt application performance if access cannot be restricted inside locality domains. This *locality problem* is the first of two obstacles to take with high performance software on ccNUMA. It occurs even if there is only one serial program running on a ccNUMA machine. The second problem is potential *contention* if two processors from different locality domains access memory in the same locality domain, fighting for memory bandwidth. Even if the network is nonblocking and its performance matches the bandwidth and latency of local access, contention can occur. Both problems can be solved by carefully observing the data access patterns of an application and restricting data access of each processor to its own locality domain. Chapter 8 will elaborate on this topic.

In inexpensive ccNUMA systems I/O interfaces are often connected to a single LD. Although I/O transfers are usually slow compared to memory bandwidth, there are, e.g., high-speed network interconnects that feature multi-GB bandwidths

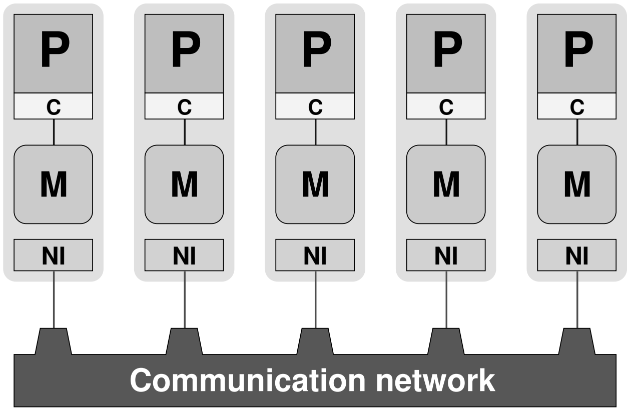


Figure 4.7: Simplified programmer's view, or "programming model," of a distributed-memory parallel computer: Separate processes run on processors (P), communicating via interfaces (NI) over some network. No process can access another process' memory (M) directly, although processors may reside in shared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

between compute nodes. If data arrives at the "wrong" locality domain, written by an I/O driver that has positioned its buffer space disregarding any ccNUMA constraints, it should be copied to its optimal destination, reducing effective bandwidth by a factor of four (three if write allocates can be avoided, see Section 1.3.1). In this case even the most expensive interconnect hardware is wasted. In truly scalable ccNUMA designs this problem is circumvented by distributing I/O connections across the whole machine and using ccNUMA-aware drivers.

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.

Each node comprises at least one network interface (NI) that mediates the connection to a *communication network*. A serial process runs on each CPU that can communicate with other processes on other CPUs by means of the network. It is easy to envision how several processors could work together on a common problem in a shared-memory parallel computer, but as there is no remote memory access on distributed-memory machines, the problem has to be solved cooperatively by sending messages back and forth between processes. Chapter 9 gives an introduction to the dominating message passing standard, MPI. Although message passing is much more

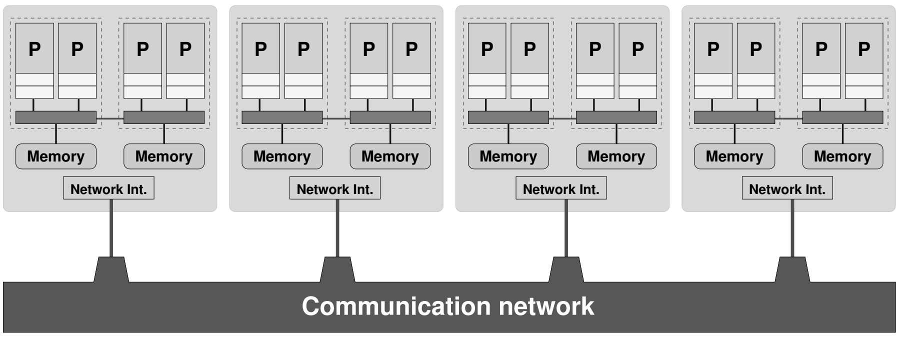


Figure 4.8: Typical hybrid system with shared-memory nodes (ccNUMA type). Two-socket building blocks represent the price vs. performance "sweet spot" and are thus found in many commodity clusters.

complex to use than any shared-memory programming paradigm, large-scale supercomputers are exclusively of the distributed-memory variant on a "global" level.

The distributed-memory architecture outlined here is also named *No Remote Memory Access* (NORMA). Some vendors provide libraries and sometimes hardware support for limited remote memory access functionality even on distributed-memory machines. Since such features are strongly vendor-specific, and there is no widely accepted standard available, a detailed coverage would be beyond the scope of this book.

There are many options for the choice of interconnect. In the simplest case one could use standard switched Ethernet, but a number of more advanced technologies have emerged that can easily have ten times the performance of Gigabit Ethernet (see Section 4.5.1 for an account of basic performance characteristics of networks). As will be shown in Section 5.3, the layout and "speed" of the network has considerable impact on application performance. The most favorable design consists of a nonblocking "wirespeed" network that can switch $N/2$ connections between its N participants without any bottlenecks. Although readily available for small systems with tens to a few hundred nodes, nonblocking switch fabrics become vastly expensive on very large installations and some compromises are usually made, i.e., there will be a bottleneck if all nodes want to communicate concurrently. See Section 4.5 for details on network topologies.

4.4 Hierarchical (hybrid) systems

As already mentioned, large-scale parallel computers are neither of the purely shared-memory nor of the purely distributed-memory type but a mixture of both, i.e., there are shared-memory building blocks connected via a fast network. This makes the overall system design even more anisotropic than with multicore processors and ccNUMA nodes, because the network adds another level of communication characteristics (see Figure 4.8). The concept has clear advantages in terms of price vs. performance; it is cheaper to build a shared-memory node with two sockets instead of two nodes with one socket each, as much of the infrastructure can be shared. Moreover, with more cores or sockets sharing a single network connection, the cost for networking is reduced.

Two-socket building blocks are currently the "sweet spot" for inexpensive *commodity clusters*, i.e., systems built from standard components that were not specifically designed for high performance computing. Depending on which applications are run on the system, this compromise may lead to performance limitations due to the reduced available network bandwidth per core. Moreover, it is *per se* unclear how the complex hierarchy of cores, cache groups, sockets and nodes can be utilized efficiently. The only general consensus is that the optimal programming model is highly application- and system-dependent. Options for programming hierarchical systems are outlined in Chapter 11.

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.

4.5 Networks

We will see in Section 5.3.6 that communication overhead can have significant impact on application performance. The characteristics of the network that connects the "execution units," "processors," "compute nodes," or whatever play a dominant role here. A large variety of network technologies and topologies are available on the market, some proprietary and some open. This section tries to shed some light on the topologies and performance aspects of the different types of networks used in high performance computing. We try to keep the discussion independent of concrete implementations or programming models, and most considerations apply to distributed-memory, shared-memory, and hierarchical systems alike.

4.5.1 Basic performance characteristics of networks

As mentioned before, there are various options for the choice of a network in a parallel computer. The simplest and cheapest solution to date is Gigabit Ethernet, which will suffice for many throughput applications but is far too slow for parallel programs with any need for fast communication. At the time of writing, the domi-

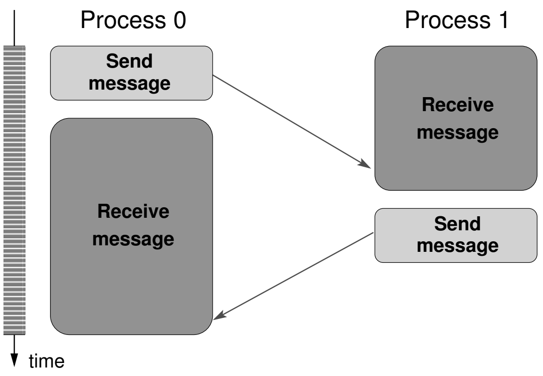


Figure 4.9: Timeline for a "Ping-Pong" data exchange between two processes. PingPong reports the time it takes for a message of length N bytes to travel from process 0 to process 1 and back.

nating distributed-memory interconnect, especially in commodity clusters, is *InfiniBand*.

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,

$$T = T_\ell + \frac{N}{B} \tag{4.1}$$

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

$$B_{\text{eff}} = \frac{N}{T_\ell + \frac{N}{B}} \ . \tag{4.2}$$

Note that in the most general case T_ℓ and B depend on the message length N. A multicore processor chip with a shared cache as shown, e.g., in Figure 1.18, is a typical example: Latency and bandwidth of message transfers between two cores on the same socket certainly depend on whether the message fits into the shared cache. We will ignore such effects for now, but they are vital to understand the finer details of message passing optimizations, which will be covered in Chapter 10.

For the measurement of latency and effective bandwidth the *PingPong* benchmark is frequently used. The code sends a message of size N [bytes] once back and forth between two processes running on different processors (and probably different nodes as well; see Figure 4.9). In pseudocode this looks as follows:

```
myID = get_process_ID()
if(myID.eq.0) then
  targetID = 1
  S = get_walltime()
  call Send_message(buffer,N,targetID)
  call Receive_message(buffer,N,targetID)
  E = get_walltime()
  MBYTES = 2*N/(E-S)/1.d6     ! MBytes/sec rate
  TIME    = (E-S)/2*1.d6       ! transfer time in microsecs
                               ! for single message
else
  targetID = 0
  call Receive_message(buffer,N,targetID)
  call Send_message(buffer,N,targetID)
endif
```

Bandwidth in MBytes/sec is then reported for different N. In reality one would use an appropriate messaging library like the Message Passing Interface (MPI), which will be introduced in Chapter 9. The data shown below was obtained using the standard "Intel MPI Benchmarks" (IMB) suite [W124].

In Figure 4.10, the model parameters in (4.2) are fitted to real data measured on a Gigabit Ethernet network. This simple model is able to describe the gross features well: We observe very low bandwidth for small message sizes, because latency dominates the transfer time. For very large messages, latency plays no role any more and effective bandwidth saturates. The fit parameters indicate plausible values for Gigabit Ethernet; however, latency can certainly be measured directly by taking the $N = 0$ limit of transfer time (inset in Figure 4.10). Obviously, the fit cannot reproduce T_ℓ accurately. See below for details.

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.

In fact, high-performance networks try to improve latency by reducing the influence of all of the above. Lightweight protocols, optimized drivers, and communication devices directly attached to processor buses are all employed by vendors to provide low latency.

One should, however, not be overly confident of the quality of fits to the model (4.2). After all, the message sizes vary across eight orders of magnitude, and the effective bandwidth in the latency-dominated regime is at least three orders of magnitude smaller than for large messages. Moreover, the two fit parameters T_ℓ and B are relevant on different ends of the fit region. The determination of Gigabit Ethernet latency from PingPong data in Figure 4.10 failed for these reasons. Hence, it is a good idea to check the applicability of the model by trying to establish "good" fits

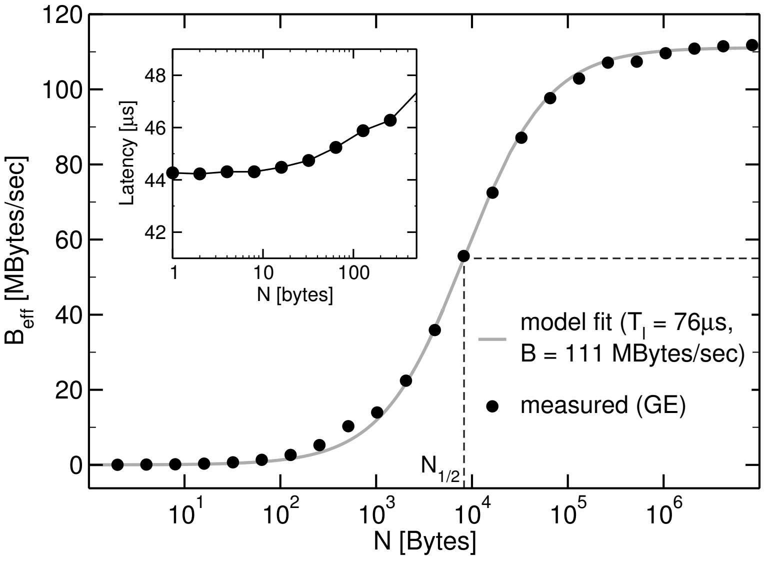


Figure 4.10: Fit of the model for effective bandwidth (4.2) to data measured on a GigE network. The fit cannot accurately reproduce the measured value of T_ℓ (see text). $N_{1/2}$ is the message length at which half of the saturation bandwidth is reached (dashed line).

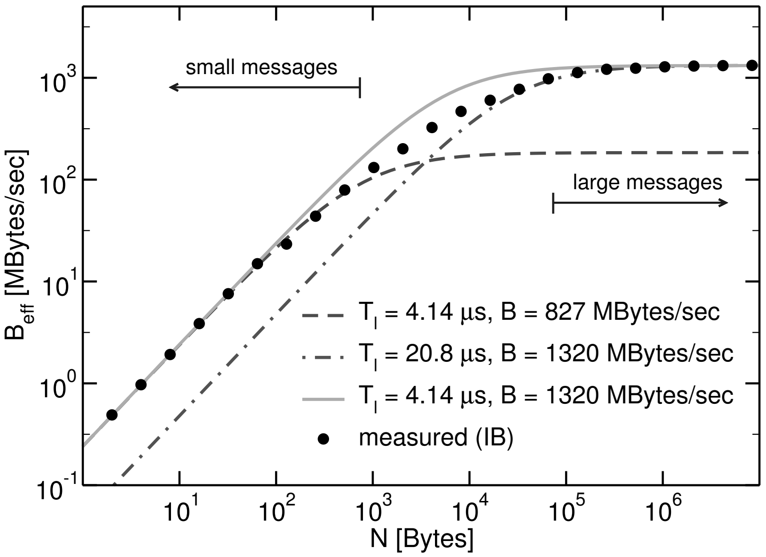


Figure 4.11: Fits of the model for effective bandwidth (4.2) to data measured on a DDR InfiniBand network. “Good” fits for asymptotic bandwidth (dotted-dashed) and latency (dashed) are shown separately, together with a fit function that unifies both (solid).

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; common effects are that the message-passing or network protocol layers switch between different buffering algorithms at a certain message size (see also Section 10.2), or that messages have to be split into smaller chunks as they become larger than some limit.

Although the saturation bandwidth B can be quite high (there are systems where the achievable internode network bandwidth is comparable to the local memory bandwidth of the processor), many applications work in a region on the bandwidth graph where latency effects still play a dominant role. To quantify this problem, the $N_{1/2}$ value is often reported. This is the message size at which $B_\text{eff} = B/2$ (see Figure 4.10). In the model (4.2), $N_{1/2} = BT_\ell$. From this point of view it makes sense to ask whether an increase in maximum network bandwidth by a factor of β is really beneficial for all messages. At message size N, the improvement in effective bandwidth is

$$\frac{B_\text{eff}(\beta B, T_\ell)}{B_\text{eff}(B, T_\ell)} = \frac{1 + N/N_{1/2}}{1 + N/\beta N_{1/2}} \, , \tag{4.3}$$

so that for $N = N_{1/2}$ and $\beta = 2$ the gain is only 33%. In case of a reduction of latency by a factor of β, the result is the same. Thus, it is desirable to improve on both latency and bandwidth to make an interconnect more efficient for all applications.

Bisection bandwidth

Note that the simple PingPong algorithm described above cannot pinpoint "global" saturation effects: If the network fabric is not completely nonblocking and all nodes transmit or receive data at the same time, aggregated bandwidth, i.e., the sum over all effective bandwidths for all point-to-point connections, is lower than the theoretical limit. This can severely throttle the performance of applications on large CPU numbers as well as overall throughput of the machine. One helpful metric to quantify the maximum aggregated communication capacity across the whole network is its *bisection bandwidth* B_b. It is the sum of the bandwidths of the minimal number of connections cut when splitting the system into two equal-sized parts (dashed line in Figure 4.12). In hybrid/hierarchical systems, a more meaningful metric is actually the available bandwidth per core, i.e., bisection bandwidth divided by the overall number of compute cores. It is one additional adverse effect of the multicore transition that bisection bandwidth per core goes down.

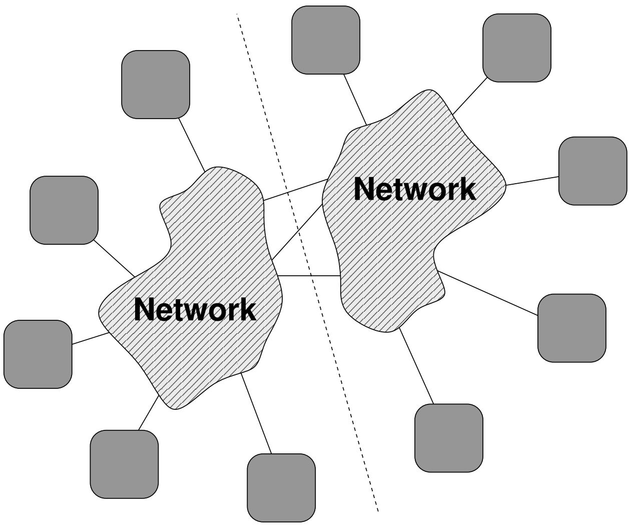


Figure 4.12: The bisection bandwidth B_b is the sum of the bandwidths of the minimal number of connections cut (three in this example) when dividing the system into two equal parts.

4.5.2 Buses

A bus is a shared medium that can be used by exactly one communicating device at a time (Figure 4.13). Some appropriate hardware mechanism must be present that detects collisions (i.e., attempts by two or more devices to transmit concurrently). Buses are very common in computer systems. They are easy to implement, feature lowest latency at small utilization, and ready-made hardware components are available that take care of the necessary protocols. A typical example is the PCI (Peripheral Component Interconnect) bus, which is used in many commodity systems to connect I/O components. In some current multicore designs, a bus connects separate CPU chips in a common package with main memory.

The most important drawback of a bus is that it is *blocking*. All devices share a constant bandwidth, which means that the more devices are connected, the lower the average available bandwidth per device. Moreover, it is technically involved to design fast buses for large systems as capacitive and inductive loads limit transmission speeds. And finally, buses are susceptible to failures because a local problem can easily influence all devices. In high performance computing the use of buses for high-speed communication is usually limited to the processor or socket level, or to diagnostic networks.

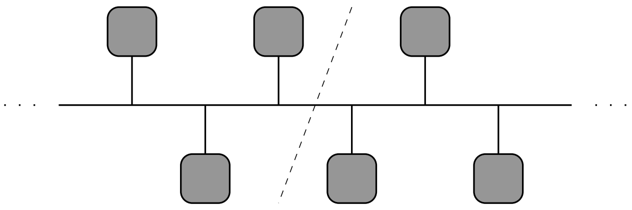


Figure 4.13: A bus network (shared medium). Only one device can use the bus at any time, and bisection bandwidth is independent of the number of nodes.

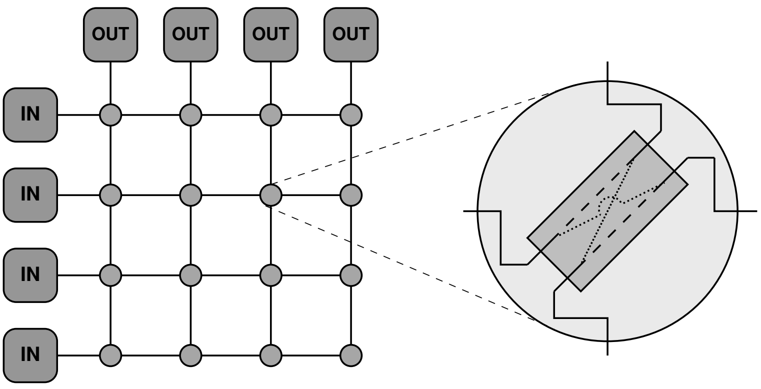


Figure 4.14: A flat, fully nonblocking two-dimensional crossbar network. Each circle represents a possible connection between two devices from the "IN" and "OUT" groups, respectively, and is implemented as a 2×2 switching element. The whole circuit can act as a four-port nonblocking switch.

4.5.3 Switched and fat-tree networks

A switched network subdivides all communicating devices into groups. The devices in one group are all connected to a central network entity called a *switch* in a star-like manner. Switches are then connected with each other or using additional switch layers. In such a network, the distance between two communicating devices varies according to how many "hops" a message has to sustain before it reaches its destination. Therefore, a multiswitch hierarchy is necessarily heterogeneous with respect to latency. The maximum number of hops required to connect two arbitrary devices is called the *diameter* of the network. For a bus (see Section 4.5.2), the diameter is one.

A single switch can either support a fully *nonblocking* operation, which means that all pairs of ports can use their full bandwidth concurrently, or it can have — partly or completely — a bus-like design where bandwidth is limited. One possible implementation of a fully nonblocking switch is a *crossbar* (see Figure 4.14). Such building blocks can be combined and cascaded to form a *fat tree* switch hierarchy, leaving a choice as to whether to keep the nonblocking property across the whole system (see Figure 4.15) or to tailor available bandwidth by using thinner connections towards the root of the tree (see Figure 4.16). In this case the bisection bandwidth per compute element is less than half the leaf switch bandwidth per port, and contention will occur even if static routing is itself not a problem. Note that the network infrastructure must be capable of (dynamically or statically) balancing the traffic from all the leaves over the thinly populated higher-level connections. If this is not possible, some node-to-node connections may be faster than others even if the network is lightly loaded. On the other hand, maximum latency between two arbitrary compute elements usually depends on the number of switch hierarchy layers only.

Compromises of the latter kind are very common in very large systems as estab-

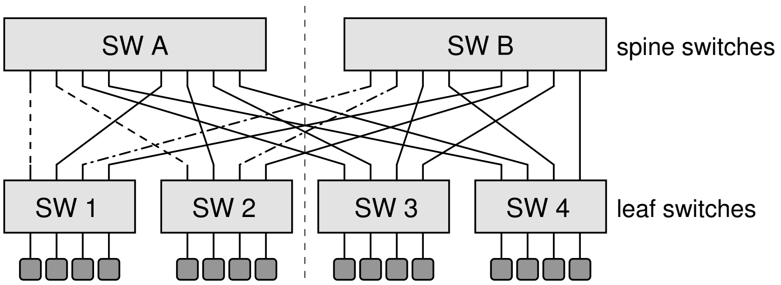


Figure 4.15: A fully nonblocking full-bandwidth fat-tree network with two switch layers. The switches connected to the actual compute elements are called *leaf switches*, whereas the upper layers form the *spines* of the hierarchy.

lishing a fully nonblocking switch hierarchy across thousands of compute elements becomes prohibitively expensive and the required hardware for switches and cabling gets easily out of hand. Additionally, the network turns heterogeneous with respect to available aggregated bandwidth — depending on the actual communication requirements of an application it may be crucial for overall performance where exactly the workers are located across the system: If a group of workers use a single leaf switch, they might enjoy fully nonblocking communication regardless of the bottleneck further up (see Section 4.5.4 for alternative approaches to build very large high-performance networks that try avoid this kind of problem).

There may however still exist bottlenecks even with a fully nonblocking switch hierarchy like the one shown in Figure 4.15. If *static routing* is used, i.e., if connections between compute elements are "hardwired" in the sense that there is one and only one chosen data path (sequence of switches traversed) between any two, one can easily encounter situations where the utilization of spine switch ports is unbal-

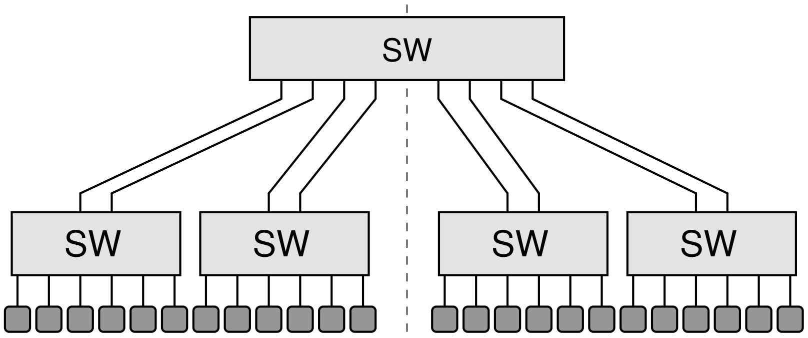


Figure 4.16: A fat-tree network with a bottleneck due to "1:3 oversubscription" of communication links to the spine. By using a single spine switch, the bisection bandwidth is cut in half as compared to the layout in Figure 4.15 because only four nonblocking pairs of connections are possible. Bisection bandwidth per compute element is even lower.

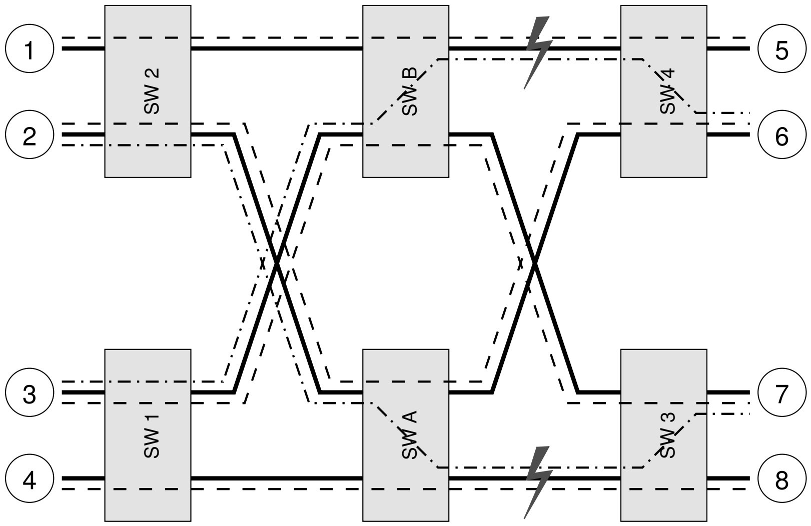


Figure 4.17: Even in a fully nonblocking fat-tree switch hierarchy (network cabling shown as solid lines), not all possible combinations of $N/2$ point-to-point connections allow collision-free operation under static routing. When, starting from the collision-free connection pattern shown with dashed lines, the connections 2↔6 and 3↔7 are changed to 2↔7 and 3↔6, respectively (dotted-dashed lines), collisions occur, e.g., on the highlighted links *if* connections 1↔5 and 4↔8 are not re-routed at the same time.

anced, leading to collisions when the load is high (see Figure 4.17 for an example). Many commodity switch products today use static routing tables [O57]. In contrast, *adaptive routing* selects data paths depending on the network load and thus avoids collisions. Only adaptive routing bears the potential of making full use of the available bisection bandwidth for all communication patterns.

4.5.4 Mesh networks

Fat-tree switch hierarchies have the disadvantage of limited scalability in very large systems, mostly in terms of price vs. performance. The cost of active components and the vast amount of cabling are prohibitive and often force compromises like the reduction of bisection bandwidth per compute element. In order to overcome those drawbacks and still arrive at a controllable scaling of bisection bandwidth, large MPP machines like the IBM Blue Gene [V114, V115, V116] or the Cray XT [V117] use *mesh networks*, usually in the form of multidimensional (hyper-)cubes. Each compute element is located at a Cartesian grid intersection. Usually the connections are wrapped around the boundaries of the hypercube to form a torus topology (see Figure 4.18 for a 2D torus example). There are no direct connections between elements that are not next neighbors. The task of routing data through the system is usually accomplished by special ASICs (application specific integrated circuits), which

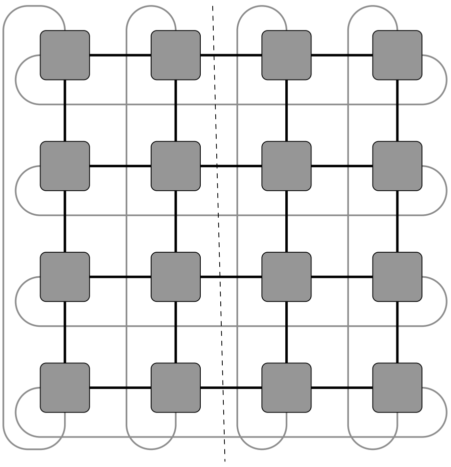


Figure 4.18: A two-dimensional (square) torus network. Bisection bandwidth scales like $\sqrt{N}$ in this case.

take care of all network traffic, bypassing the CPU whenever possible. The network diameter is the sum of the system's sizes in all three Cartesian directions.

Certainly, bisection bandwidth does not scale linearly when enlarging the system in all dimensions but behaves like $B_\mathrm{b}(N) \propto N^{(d-1)/d}$ (d being the number of dimensions), which leads to $B_\mathrm{b}(N)/N \to 0$ for large N. Maximum latency scales like $N^{1/d}$. Although these properties appear unfavorable at first sight, the torus topology is an acceptable and extremely cost-effective compromise for the large class of applications that are dominated by nearest-neighbor communication. If the maximum bandwidth per link is substantially larger than what a single compute element can "feed" into the network (its *injection bandwidth*), there is enough headroom to support more demanding communication patterns as well (this is the case, for instance, on the Cray XT line of massively parallel computers [V117]). Another advantage of a cubic mesh is that the amount of cabling is limited, and most cables can be kept short. As with fat-tree networks, there is some heterogeneity in bandwidth and latency behavior, but if compute elements that work in parallel to solve a problem are located close together (i.e., in a cuboidal region), these characteristics are well predictable. Moreover, there is no "arbitrary" system size at which bisection bandwidth per node suddenly has to drop due to cost and manageability concerns.

On smaller scales, simple mesh networks are used in shared-memory systems for ccNUMA-capable connections between locality domains. Figure 4.19 shows an example of a four-socket server with HyperTransport interconnect. This node actually implements a heterogeneous topology (in terms of intersocket latency) because two HT connections are used for I/O connectivity: Any communication between the two locality domains on the right incurs an additional hop via one of the other domains.

4.5.5 Hybrids

If a network is built as a combination of at least two of the topologies described above, it is called *hybrid*. In a sense, a cluster of shared-memory nodes like in Fig-

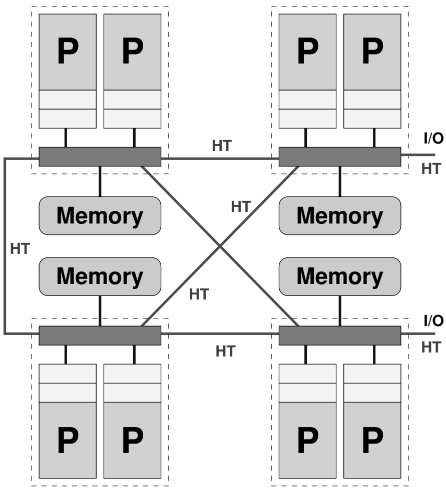


Figure 4.19: A four-socket ccNUMA system with a HyperTransport-based mesh network. Each socket has only three HT links, so the network has to be heterogeneous in order to accommodate I/O connections and still utilize all provided HT ports.

ure 4.8 implements a hybrid network even if the internode network itself is not hybrid. This is because intranode connections tend to be buses (in multicore chips) or simple meshes (for ccNUMA-capable fabrics like HyperTransport or QuickPath). On the large scale, using a cubic topology for node groups of limited size and a non-blocking fat tree further up reduces the bisection bandwidth problems of pure cubic meshes.

Problems

For solutions see page 295 *ff.*

4.1 *Building fat-tree network hierarchies.* In a fat-tree network hierarchy with static routing, what are the consequences of a 2:3 oversubscription on the links to the spine switches?