

# Chapter 3

---

## Data access optimization

Of all possible performance-limiting factors in HPC, the most important one is data access. As explained earlier, microprocessors tend to be inherently “unbalanced” with respect to the relation of theoretical peak performance versus memory bandwidth. Since many applications in science and engineering consist of loop-based code that moves large amounts of data in and out of the CPU, on-chip resources tend to be underutilized and performance is limited only by the relatively slow data paths to memory or even disks.

Figure 3.1 shows an overview of several data paths present in modern parallel computer systems, and typical ranges for their bandwidths and latencies. The functional units, which actually perform the computational work, sit at the top of this hierarchy. In terms of bandwidth, the slowest data paths are three to four orders of magnitude away, and eight in terms of latency. The deeper a data transfer must reach down through the different levels in order to obtain required operands for some calculation, the harder the impact on performance. Any optimization attempt should therefore first aim at reducing traffic over slow data paths, or, should this turn out to be infeasible, at least make data transfer as efficient as possible.

---

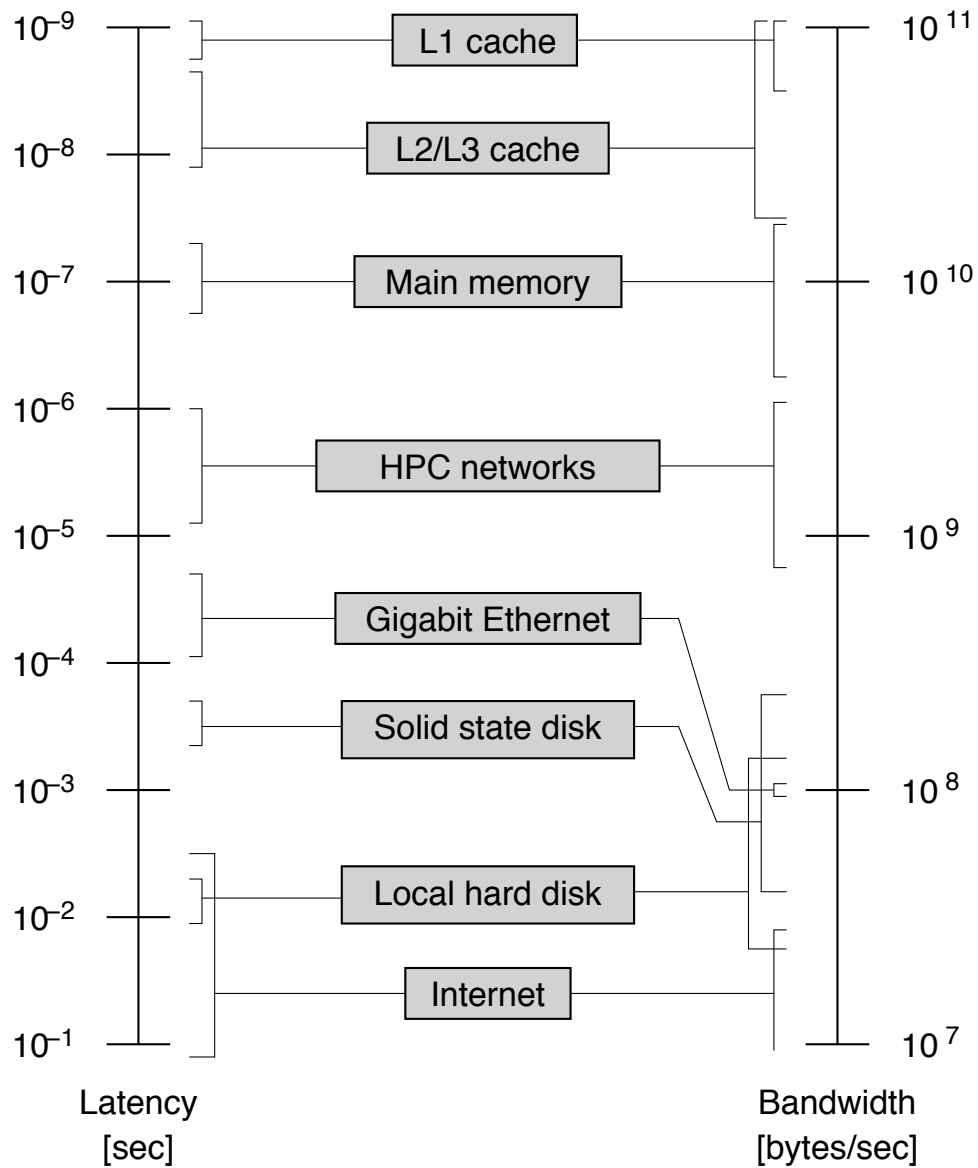
### 3.1 Balance analysis and lightspeed estimates

#### 3.1.1 Bandwidth-based performance modeling

Some programmers go to great lengths trying to improve the efficiency of code. In order to decide whether this makes sense or if the program at hand is already using the resources in the best possible way, one can often estimate the theoretical performance of loop-based code that is bound by bandwidth limitations by simple rules of thumb. The central concept to introduce here is *balance*. For example, the *machine balance*  $B_m$  of a processor chip is the ratio of possible memory bandwidth in GWords/sec to peak performance in GFlops/sec:

$$B_m = \frac{\text{memory bandwidth [GWords/sec]}}{\text{peak performance [GFlops/sec]}} = \frac{b_{\max}}{P_{\max}} \quad (3.1)$$

“Memory bandwidth” could also be substituted by the bandwidth to caches or even network bandwidths, although the metric is generally most useful for codes that are really memory-bound. Access latency is assumed to be hidden by techniques like



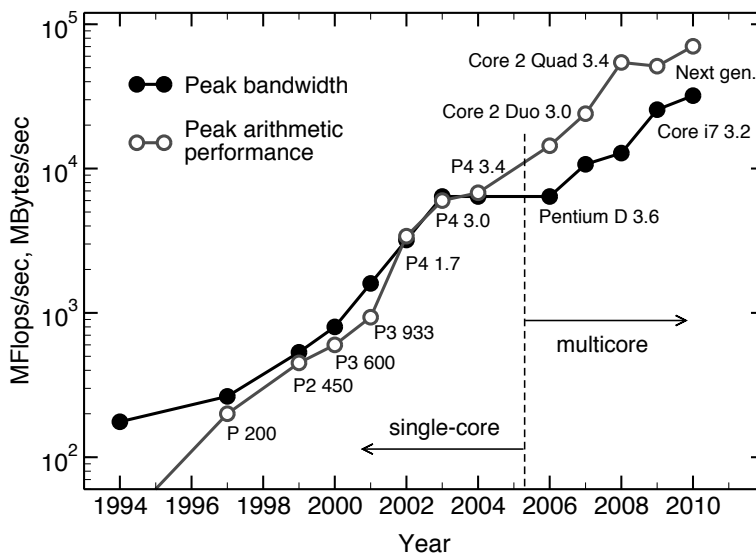
**Figure 3.1:** Typical latency and bandwidth numbers for data transfer to and from different devices in computer systems. Registers have been omitted because their “bandwidth” usually matches the computational capabilities of the compute core, and their latency is part of the pipelined execution.

data path	balance [W/F]
cache	0.5–1.0
<b>machine (memory)</b>	0.03–0.5
interconnect (high speed)	0.001–0.02
interconnect (Gbit ethernet)	0.0001–0.0007
disk (or disk subsystem)	0.0001–0.01

**Table 3.1:** Typical balance values for operations limited by different transfer paths. In case of network and disk connections, the peak performance of typical dual-socket compute nodes was taken as a basis.

prefetching and software pipelining. As an example, consider a dual-core chip with a clock frequency of 3.0 GHz that can perform at most four flops per cycle (per core) and has a memory bandwidth of 10.6 GBytes/sec. This processor would have a machine balance of 0.055 W/F. At the time of writing, typical values of  $B_m$  lie in the range between 0.03 W/F for standard cache-based microprocessors and 0.5 W/F for top of the line vector processors. Due to the continuously growing DRAM gap and the increasing core counts, machine balance for standard architectures will presumably decrease further in the future. Table 3.1 shows typical balance values for several different transfer paths.

In Figure 3.2 we have collected peak performance and memory bandwidth data for Intel processors between 1994 and 2010. Each year the desktop processor with the fastest clock speed was chosen as a representative. Although peak performance did grow faster than memory bandwidth before 2005, the introduction of the first dual-core chip (Pentium D) really widened the DRAM gap considerably. The Core i7 design gained some ground in terms of bandwidth, but the long-term general trend is clearly unperturbed by such exceptions.



**Figure 3.2:** Progress of maximum arithmetic performance (open circles) and peak theoretical memory bandwidth (filled circles) for Intel processors since 1994. The fastest processor in terms of clock frequency is shown for each year. (Data collected by Jan Treibig.)

In order to quantify the requirements of some code that runs on a machine with a certain balance, we further define the *code balance* of a loop to be

$$B_c = \frac{\text{data traffic [Words]}}{\text{floating point ops [Flops]}} . \quad (3.2)$$

“Data traffic” refers to all words transferred over the performance-limiting data path, which makes this metric to some extent dependent on the hardware (see Section 3.3 for an example). The reciprocal of code balance is often called *computational intensity*. We can now calculate the expected maximum fraction of peak performance of a code with balance  $B_c$  on a machine with balance  $B_m$ :

$$l = \min \left( 1, \frac{B_m}{B_c} \right) . \quad (3.3)$$

We call this fraction the *lightspeed* of a loop. Performance in GFlops/sec is then

$$P = lP_{\max} = \min \left( P_{\max}, \frac{b_{\max}}{B_c} \right) \quad (3.4)$$

If  $l \simeq 1$ , performance is not limited by bandwidth but other factors, either inside the CPU or elsewhere. Note that this simple performance model is based on some crucial assumptions:

- The loop code makes use of all arithmetic units (MULT and ADD) in an optimal way. If this is not the case one must introduce a correction factor that reflects the ratio of “effective” to absolute peak performance (e.g., if only ADDs are used in the code, effective peak performance would be half of the absolute maximum). Similar considerations apply if less than the maximum number of cores per chip are used.
- The loop code is based on double precision floating-point arithmetic. However, one can easily derive similar metrics that are more appropriate for other codes (e.g., 32-bit words per integer arithmetic instruction etc.).
- Data transfer and arithmetic overlap perfectly.
- The slowest data path determines the loop code’s performance. All faster data paths are assumed to be infinitely fast.
- The system is in “throughput mode,” i.e., latency effects are negligible.
- It is possible to saturate the memory bandwidth that goes into the calculation of machine balance to its full extent. Recent multicore designs tend to underutilize the memory interface if only a fraction of the cores use it. This makes performance prediction more complex, since there is a separate “effective” machine balance that is not just proportional to  $N^{-1}$  for each core count  $N$ . See Section 3.1.2 and Problem 3.1 below for more discussion regarding this point.

type	kernel	DP words	flops	$B_c$
COPY	$A(:) = B(:)$	2 (3)	0	N/A
SCALE	$A(:) = s * B(:)$	2 (3)	1	2.0 (3.0)
ADD	$A(:) = B(:) + C(:)$	3 (4)	1	3.0 (4.0)
TRIAD	$A(:) = B(:) + s * C(:)$	3 (4)	2	1.5 (2.0)

**Table 3.2:** The STREAM benchmark kernels with their respective data transfer volumes (third column) and floating-point operations (fourth column) per iteration. Numbers in brackets take write allocates into account.

While the balance model is often useful and accurate enough to estimate the performance of loop codes and get guidelines for optimization (especially if combined with visualizations like the roofline diagram [M42]), we must emphasize that more advanced strategies for performance modeling do exist and refer to the literature [L76, M43, M41].

As an example consider the standard vector triad benchmark introduced in Section 1.3. The kernel loop,

---

```

1 do i=1,N
2   A(i) = B(i) + C(i) * D(i)
3 enddo

```

---

executes two flops per iteration, for which three loads (to elements  $B(i)$ ,  $C(i)$ , and  $D(i)$ ) and one store operation (to  $A(i)$ ) provide the required input data. The code balance is thus  $B_c = (3 + 1)/2 = 2$ . On a CPU with machine balance  $B_m = 0.1$ , we can then expect a lightspeed ratio of 0.05, i.e., 5% of peak.

Standard cache-based microprocessors usually have an outermost cache level with write-back strategy. As explained in Section 1.3, a cache line write allocate is then required after a store miss to ensure cache-memory coherence if nontemporal stores or cache line zero is not used. Under such conditions, the store stream to array  $A$  must be counted twice in calculating the code balance, and we would end up with a lightspeed estimate of  $l_{wa} = 0.04$ .

### 3.1.2 The STREAM benchmarks

The McCalpin STREAM benchmarks [134, W119] is a collection of four simple synthetic kernel loops which are supposed to fathom the capabilities of a processor's or a system's memory interface. Table 3.2 lists those operations with their respective code balance. Performance is usually reported as bandwidth in GBytes/sec. The STREAM TRIAD kernel is not to be confused with the vector triad (see previous section), which has one additional load stream.

The benchmarks exist in serial and OpenMP-parallel (see Chapter 6) variants and are usually run with data sets large enough so that performance is safely memory-bound. Measured bandwidth thus depends on the number of load and store streams only, and the results for COPY and SCALE (and likewise for ADD and TRIAD)

tend to be very similar. One must be aware that STREAM is not only defined via the loop kernels in Table 3.2, but also by its Fortran source code (there is also a C variant available). This is important because optimizing compilers can recognize the STREAM source and substitute the kernels by hand-tuned machine code. Therefore, it is safe to state that STREAM performance results reflect the true capabilities of the hardware. They are published for many historical and contemporary systems on the STREAM Web site [W119].

Unfortunately, STREAM as well as the vector triad often fail to reach the performance levels predicted by balance analysis, in particular on commodity (PC-based) hardware. The reasons for this failure are manifold and cannot be discussed here in full detail; typical factors are:

- Maximum bandwidth is often not available in both directions (read and write) concurrently. It may be the case, e.g., that the relation from maximum read to maximum write bandwidth is 2:1. A write stream cannot utilize the full bandwidth in that case.
- Protocol overhead (see, e.g., Section 4.2.1), deficiencies in chipsets, error-correcting memory chips, and large latencies (that cannot be hidden completely by prefetching) all cut on available bandwidth.
- Data paths inside the processor chip, e.g., connections between L1 cache and registers, can be unidirectional. If the code is not balanced between read and write operations, some of the bandwidth in one direction is unused. This should be taken into account when applying balance analysis for in-cache situations.

It is, however, still true that STREAM results mark a maximum for memory bandwidth and no real application code with similar characteristics (number of load and store streams) performs significantly better. Thus, the STREAM bandwidth  $b_S$  rather than the hardware's theoretical capabilities should be used as the reference for light-speed calculations and (3.4) be modified to read

$$P = \min \left( P_{\max}, \frac{b_S}{B_c} \right) \quad (3.5)$$

Getting a significant fraction (i.e., 80% or more) of the predicted performance based on STREAM results for an application code is usually an indication that there is no more potential for improving the utilization of the memory interface. It does not mean, however, that there is no room for further optimizations. See the following sections.

As an example we pick a system with Intel's Xeon 5160 processor (see Figure 4.4 for the general layout). One core has a theoretical memory bandwidth of  $b_{\max} = 10.66$  GBytes/sec and a peak performance of  $P_{\max} = 12$  GFlops/sec (4 flops per cycle at 3.0 GHz). This leads to a machine balance of  $B_m = 0.111$  W/F for a single core (if both cores run memory-bound code, this is reduced by a factor of two, but we assume for now that only one thread is running on one socket of the system).

Table 3.3 shows the STREAM results on this platform, comparing versions with

type	with write allocate			w/o write allocate	
	reported	actual	$b_S/b_{\max}$	reported	$b_S/b_{\max}$
COPY	2698	4047	0.38	4089	0.38
SCALE	2695	4043	0.38	4106	0.39
ADD	2772	3696	0.35	3735	0.35
TRIAD	2879	3839	0.36	3786	0.36

**Table 3.3:** Single-thread STREAM bandwidth results in GBytes/sec for an Intel Xeon 5160 processor (see text for details), comparing versions with and without write allocate. Write allocate was avoided by using nontemporal store instructions.

and without write allocate. The benchmark itself does not take this detail into account at all, so reported bandwidth numbers differ from actual memory traffic if write allocates are present. The discrepancy between measured performance and theoretical maximum is very pronounced; it is generally not possible to get more than 40% of peak bandwidth on this platform, and efficiency is particularly low for ADD and TRIAD, which have two load streams instead of one. If these results are used as a reference for balance analysis of loops, COPY or SCALE should be used in load/store-balanced cases. See Section 3.3 for an example.

## 3.2 Storage order

Multidimensional arrays, first and foremost matrices or matrix-like structures, are omnipresent in scientific computing. Data access is a crucial topic here as the mapping between the inherently one-dimensional, cache line based memory layout of standard computers and any multidimensional data structure must be matched to the order in which code loads and stores data so that spatial and temporal locality can be employed. *Strided* access to a one-dimensional array reduces spatial locality, leading to low utilization of the available bandwidth (see also Problem 3.1). When dealing with multidimensional arrays, those access patterns can be generated quite naturally:

### Stride-N access

```

1 do i=1,N
2   do j=1,N
3     A(i,j) = i*j
4   enddo
5 enddo
```

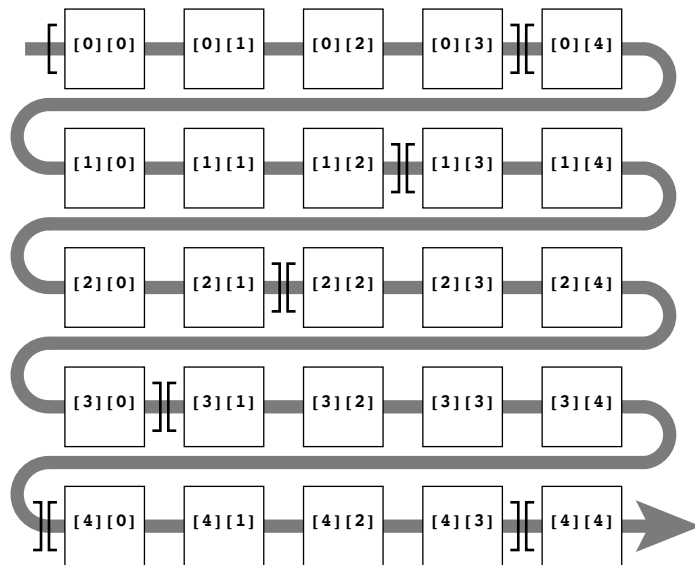
### Stride-1 access

```

for(i=0; i<N; ++i) {
  for(j=0; j<N; ++j) {
    a[i][j] = i*j;
  }
}
```

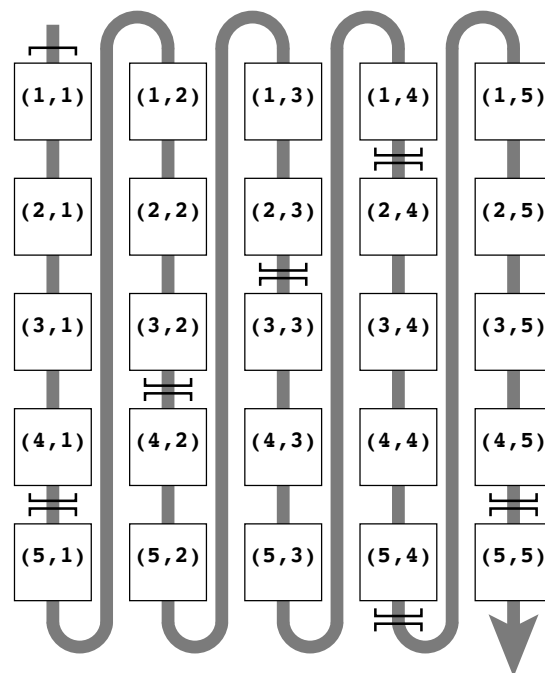
These Fortran and C codes perform exactly the same task, and the second array index is the “fast” (inner loop) index both times, but the memory access patterns are

**Figure 3.3:** Row major order matrix storage scheme, as used by the C programming language. Matrix rows are stored consecutively in memory. Cache lines are assumed to hold four matrix elements and are indicated by brackets.



quite distinct. In the Fortran example, the memory address is incremented in steps of `N*sizeof(double)`, whereas in the C example the stride is optimal. This is because C implements *row major order* (see Figure 3.3), whereas Fortran follows the so-called *column major order* (see Figure 3.4) for multidimensional arrays. Although mathematically insignificant, the distinction must be kept in mind when optimizing for data access: If an inner loop variable is used as an index to a multidimensional array, it should be the index that ensures stride-one access (i.e., the first in Fortran and the last in C). Section 3.4 will show what can be done if this is not easily possible.

**Figure 3.4:** Column major order matrix storage scheme, as used by the Fortran programming language. Matrix columns are stored consecutively in memory. Cache lines are assumed to hold four matrix elements and are indicated by brackets.





### 3.3 Case study: The Jacobi algorithm

The Jacobi method is prototypical for many stencil-based iterative methods in numerical analysis and simulation. In its most straightforward form, it can be used for solving the diffusion equation for a scalar function  $\Phi(\vec{r}, t)$ ,

$$\frac{\partial \Phi}{\partial t} = \Delta \Phi, \quad (3.6)$$

on a rectangular lattice subject to Dirichlet boundary conditions. The differential operators are discretized using finite differences (we restrict ourselves to two dimensions with no loss of generality, but see Problem 3.4 for how 2D and 3D versions can differ with respect to performance):

$$\begin{aligned} \frac{\delta \Phi(x_i, y_i)}{\delta t} = & \frac{\Phi(x_{i+1}, y_i) + \Phi(x_{i-1}, y_i) - 2\Phi(x_i, y_i)}{(\delta x)^2} \\ & + \frac{\Phi(x_i, y_{i+1}) + \Phi(x_i, y_{i-1}) - 2\Phi(x_i, y_i)}{(\delta y)^2}. \end{aligned} \quad (3.7)$$

In each time step, a correction  $\delta \Phi$  to  $\Phi$  at coordinate  $(x_i, y_i)$  is calculated by (3.7) using the “old” values from the four next neighbor points. Of course, the updated  $\Phi$  values must be written to a second array. After all points have been updated (a “sweep”), the algorithm is repeated. Listing 3.1 shows a possible kernel implementation on an isotropic lattice. It “solves” for the steady state but lacks a convergence criterion, which is of no interest here. (Note that exchanging the  $\mathfrak{t}0$  and  $\mathfrak{t}1$  lattices does not have to be done element by element; compared to a naïve implementation we already gain roughly a factor of two in performance by exchanging the third array index only.)

Many optimizations are possible for speeding up this code. We will first predict its performance using balance analysis and compare with actual measurements. Figure 3.5 illustrates one five-point stencil update in the two-dimensional Jacobi algorithm. Four loads and one store are required per update, but the “downstream neighbor”  $\text{phi}(\mathfrak{i}+1, \mathfrak{k}, \mathfrak{t}0)$  is definitely used again from cache two iterations later, so only three of the four loads count for the code balance  $B_c = 1.0 \text{ W/F}$  (1.25 W/F including write allocate). However, as Figure 3.5 shows, the row-wise traversal of the lattice brings the stencil site with the largest  $k$  coordinate (i.e.,  $\text{phi}(\mathfrak{i}, \mathfrak{k}+1, \mathfrak{t}0)$ ) into the cache for the first time (we are ignoring the cache line concept for the moment). A memory transfer cannot be avoided for this value, but it will stay in the cache for three successive row traversals *if* the cache is large enough to hold more than two lattice rows. Under this condition we can assume that loading the neighbors at rows  $k$  and  $k-1$  comes at no cost, and code balance is reduced to  $B_c = 0.5 \text{ W/F}$  (0.75 W/F including write allocate). If the inner lattice dimension is gradually made larger, one, and eventually three additional loads must be satisfied from memory, leading back to the unpleasant value of  $B_c = 1.0$  (1.25) W/F.

**Listing 3.1:** Straightforward implementation of the Jacobi algorithm in two dimensions.

---

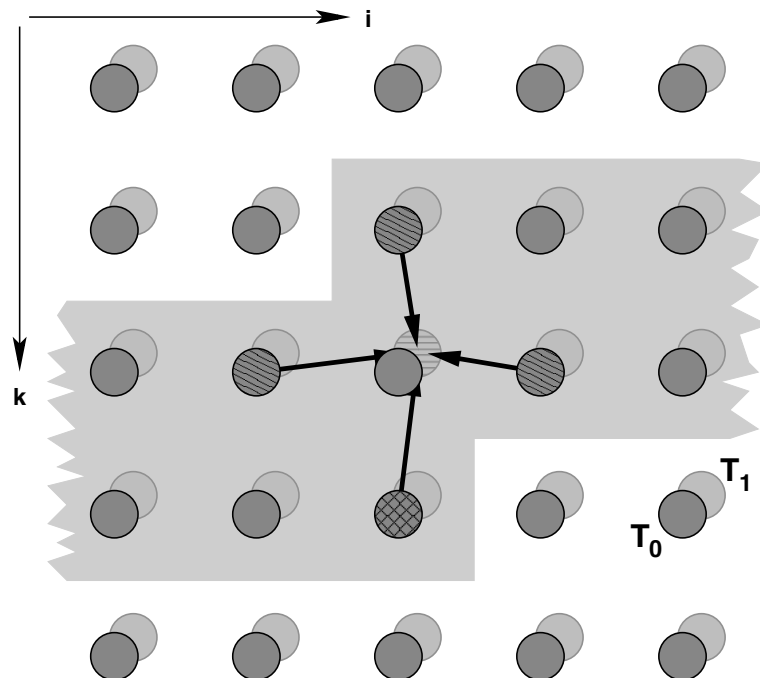
```

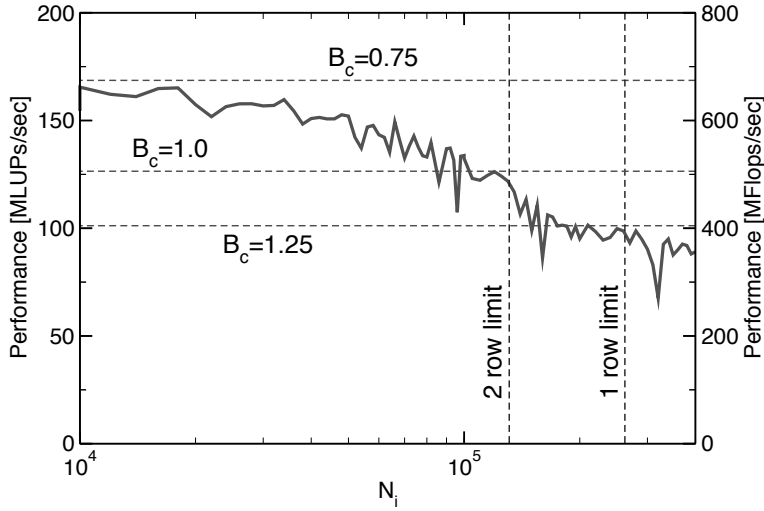
1 double precision, dimension(0:imax+1,0:kmax+1,0:1) :: phi
2 integer :: t0,t1
3 t0 = 0 ; t1 = 1
4 do it = 1,itmax      ! choose suitable number of sweeps
5   do k = 1,kmax
6     do i = 1,imax
7       ! four flops, one store, four loads
8       phi(i,k,t1) = ( phi(i+1,k,t0) + phi(i-1,k,t0)
9                       + phi(i,k+1,t0) + phi(i,k-1,t0) ) * 0.25
10    enddo
11  enddo
12  ! swap arrays
13  i = t0 ; t0=t1 ; t1=i
14 enddo

```

---

**Figure 3.5:** Stencil update for the plain 2D Jacobi algorithm. If at least two successive rows can be kept in the cache (shaded area), only one  $T_0$  site per update has to be fetched from memory (cross-hatched site).





**Figure 3.6:** Performance versus inner loop length (lattice extension in the  $i$  direction) for the Jacobi algorithm on one core of a Xeon 5160 processor (see text for details). Horizontal lines indicate predictions based on STREAM bandwidth.

Based on these balance numbers we can now calculate the code’s lightspeed on a given architecture. In Section 3.1.2 we have presented STREAM results for an Intel Xeon 5160 platform which we use as a reference here. In the case where the cache is large enough to hold two successive rows, the data transfer characteristics match those for STREAM COPY or SCALE, i.e., there is one load and one store stream plus the obligatory write allocate. The theoretical value of  $B_m = 0.111 \text{ W/F}$  has to be modified because the Jacobi kernel only comprises one MULT versus three ADD operations, hence we use

$$B_m^+ = \frac{0.111}{4/6} \text{ W/F} \approx 0.167 \text{ W/F} . \quad (3.8)$$

Based on this theoretical value and assuming that write allocates cannot be avoided we arrive at

$$l_{\text{best}} = \frac{B_m^+}{B_c} = \frac{0.167}{0.75} \approx 0.222 , \quad (3.9)$$

which, at a modified theoretical peak performance of  $P_{\text{max}}^+ = 12 \cdot 4/6 \text{ GFlops/sec} = 8 \text{ GFlops/sec}$  leads to a predicted performance of  $1.78 \text{ GFlops/sec}$ . Based on the STREAM COPY numbers from Table 3.3, however, this value must be scaled down by a factor of 0.38, and we arrive at an expected performance of  $\approx 675 \text{ MFlops/sec}$ . For very large inner grid dimensions, the cache becomes too small to hold two, and eventually even one grid row and code balance first rises to  $B_c = 1.0 \text{ W/F}$ , and finally to  $B_c = 1.25 \text{ W/F}$ . Figure 3.6 shows measured performance versus inner lattice dimension, together with the various limits and predictions (a nonsquare lattice was used for the large- $N$  cases, i.e.,  $k_{\text{max}} \ll i_{\text{max}}$ , to save memory). The model can obviously describe the overall behavior well. Small-scale performance fluctuations can have a variety of causes, e.g., associativity or memory banking effects.

The figure also introduces a performance metric that is more suitable for stencil algorithms as it emphasizes “work done” over MFlops/sec: The number of lattice site updates (LUPs) per second. In our case, there is a simple 1:4 correspondence between flops and LUPs, but in general the MFlops/sec metric can vary when applying

optimizations that interfere with arithmetic, using different compilers that rearrange terms, etc., just because the number of floating point operations per stencil update changes. However, what the user is most interested in is how much *actual work* can be done in a certain amount of time. The LUPs/sec number makes all performance measurements comparable if the underlying physical problem is the same, no matter which optimizations have been applied. For example, some processors provide a *fused multiply-add* (FMA) machine instruction which performs two flops by calculating  $r = a + b \cdot c$ . Under some circumstances, FMA can boost performance because of the reduced latency per flop. Rewriting the 2D Jacobi kernel in Listing 3.1 for FMA is straightforward:

---

```

1 do k = 1,kmax
2   do i = 1,imax
3     phi(i,k,t1) = 0.25 * phi(i+1,k,t0) + 0.25 * phi(i-1,k,t0)
4                 + 0.25 * phi(i,k+1,t0) + 0.25 * phi(i,k-1,t0)
5   enddo
6 enddo

```

---

This version has seven instead of four flops; performance in MLUPs/sec will not change for memory-bound situations (it is left to the reader to prove this using balance analysis), but the MFlops/sec numbers will.

---

### 3.4 Case study: Dense matrix transpose

For the following example we assume column major order as implemented in Fortran. Calculating the transpose of a dense matrix,  $A = B^T$ , involves strided memory access to A or B, depending on how the loops are ordered. The most unfavorable way of doing the transpose is shown here:

---

```

1 do i=1,N
2   do j=1,N
3     A(i,j) = B(j,i)
4   enddo
5 enddo

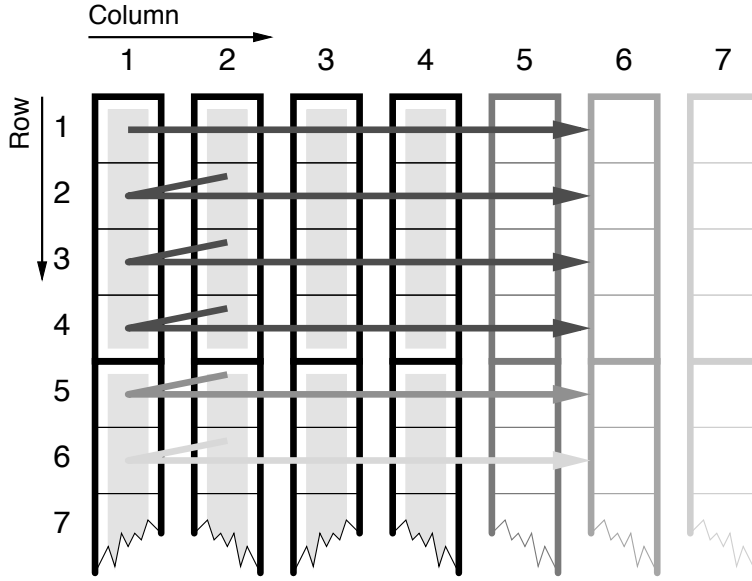
```

---

Write access to matrix A is strided (see Figure 3.7). Due to write-allocate transactions, strided writes are more expensive than strided reads. Starting from this worst possible code we can now try to derive expected performance features. As matrix transpose does not perform any arithmetic, we will use effective bandwidth (i.e., GBytes/sec available to the application) to denote performance.

Let  $C$  be the cache size and  $L_c$  the cache line size, both in DP words. Depending on the size of the matrices we can expect three primary performance regimes:

- In case the two matrices fit into a CPU cache ( $2N^2 \lesssim C$ ), we expect effective bandwidths of the order of cache speeds. Spatial locality is of importance only between different cache levels; optimization potential is limited.



**Figure 3.7:** Cache line traversal for vanilla matrix transpose (strided store stream, column major order). If the leading matrix dimension is a multiple of the cache line size, each column starts on a line boundary.

- If the matrices are too large to fit into the cache but still

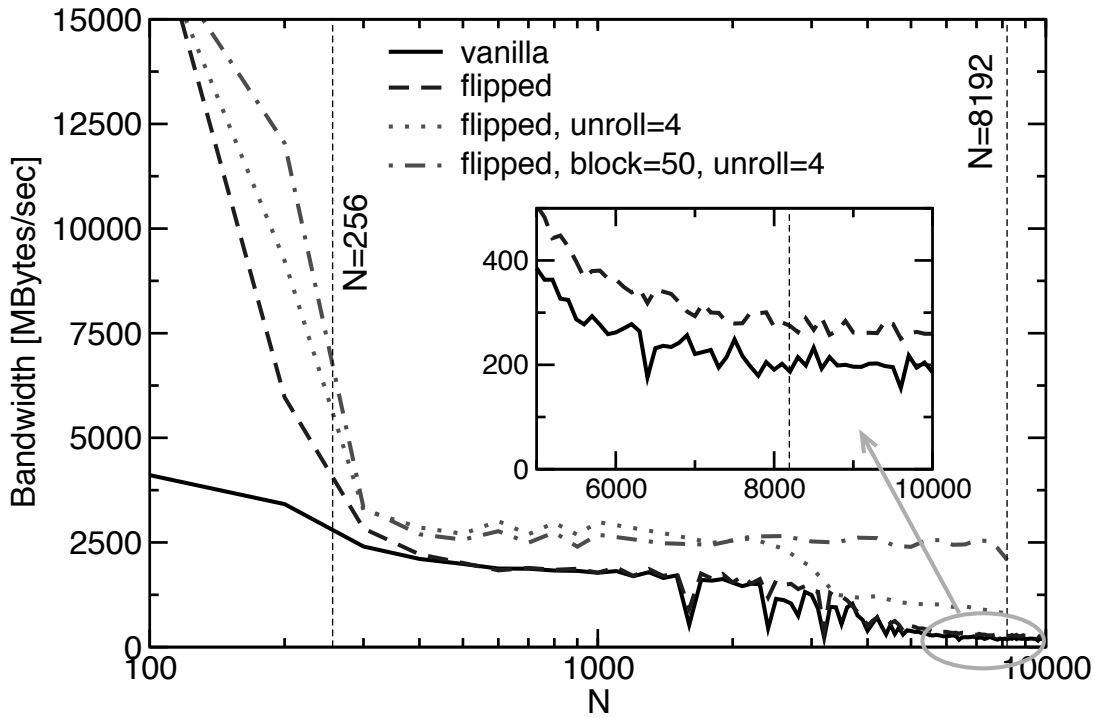
$$NL_c \lesssim C, \quad (3.10)$$

the strided access to  $A$  is insignificant because all stores that cause a write miss during a complete row traversal start a cache line write allocate. Those lines are most probably still in the cache for the next  $L_c - 1$  rows, alleviating the effect of the strided write (spatial locality). Effective bandwidth should be of the order of the processor's maximum achievable memory bandwidth.

- If  $N$  is even larger so that  $NL_c \gtrsim C$ , each store to  $A$  causes a cache miss and a subsequent write allocate. A sharp drop in performance is expected at this point as only one out of  $L_c$  cache line entries is actually used for the store stream and any spatial locality is suddenly lost.

The “vanilla” graph in Figure 3.8 shows that the assumptions described above are essentially correct, although the strided write seems to be very unfavorable even when the whole working set fits into the cache. This is probably because the L1 cache on the considered architecture (Intel Xeon/Nocona) is of *write-through* type, i.e., the L2 cache is always updated on a write, regardless of whether there was an L1 hit or miss. Hence, the write-allocate transactions between the two caches waste a major part of the available internal bandwidth.

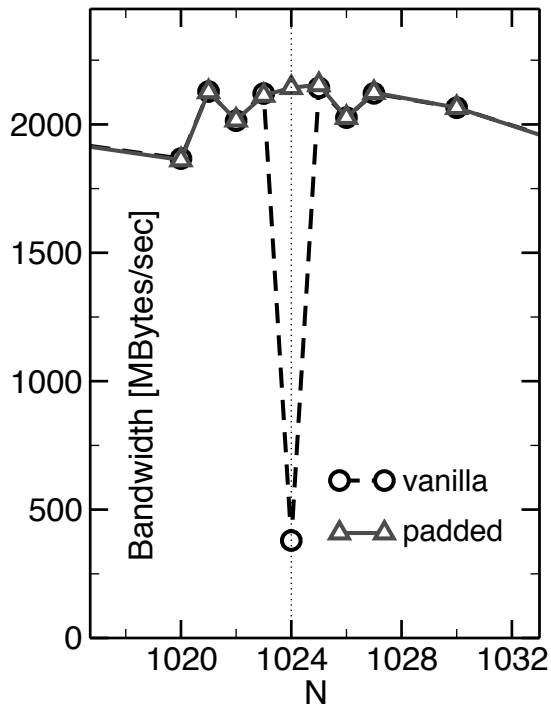
In the second regime described above, performance stays roughly constant up to a point where the fraction of cache used by the store stream for  $N$  cache lines becomes comparable to the L2 size. Effective bandwidth is around 1.8 GBytes/sec, a mediocre value compared to the theoretical maximum of 5.3 GBytes/sec (delivered by two-channel memory at 333 MTransfers/sec). On most commodity architectures the theoretical bandwidth limits can not be reached with compiler-generated code, but well over 50% is usually attainable, so there must be a factor that further reduces available bandwidth. This factor is the *translation lookaside buffer* (TLB), which



**Figure 3.8:** Performance (effective bandwidth) for different implementations of the dense matrix transpose on a modern microprocessor with 1 MByte of L2 cache. The  $N = 256$  and  $N = 8192$  lines indicate the positions where the matrices fully fit into the cache and where  $N$  cache lines fit into the cache, respectively. (Intel Xeon/Nocona 3.2 GHz.)

caches the mapping between logical and physical memory pages. The TLB can be envisioned as an additional cache level with cache lines the size of memory pages (the page size is often 4 kB, sometimes 16 kB and even configurable on some systems). On the architecture considered, it is only large enough to hold 64 entries, which corresponds to 256 kBytes of memory at a 4 kB page size. This is smaller than the whole L2 cache, so TLB effects may be observed even for in-cache situations. Moreover, if  $N$  is larger than 512, i.e., if one matrix row exceeds the size of a page, every single access in the strided stream causes a TLB *miss*. Even if the page tables reside in the L2 cache, this penalty reduces effective bandwidth significantly because every TLB miss leads to an additional access latency of at least 57 processor cycles (on this particular CPU). At a core frequency of 3.2 GHz and a bus transfer rate of 666 MWords/sec, this matches the time needed to transfer more than a 64-byte cache line!

At  $N \gtrsim 8192$ , performance has finally arrived at the expected low level. The machine under investigation has a theoretical memory bandwidth of 5.3 GBytes/sec of which around 200 MBytes/sec actually reach the application. At an effective cache line length of 128 bytes (two 64-byte cache lines are fetched on every miss, but evicted separately), of which only one is used for the strided store stream, three words per iteration are read or written in each loop iteration for the in-cache case,



**Figure 3.9:** Cache thrashing for an unfavorable choice of array dimensions (dashed): Matrix transpose performance breaks down dramatically at a dimension of  $1024 \times 1024$ . Padding by enlarging the leading dimension by one removes thrashing completely (solid).

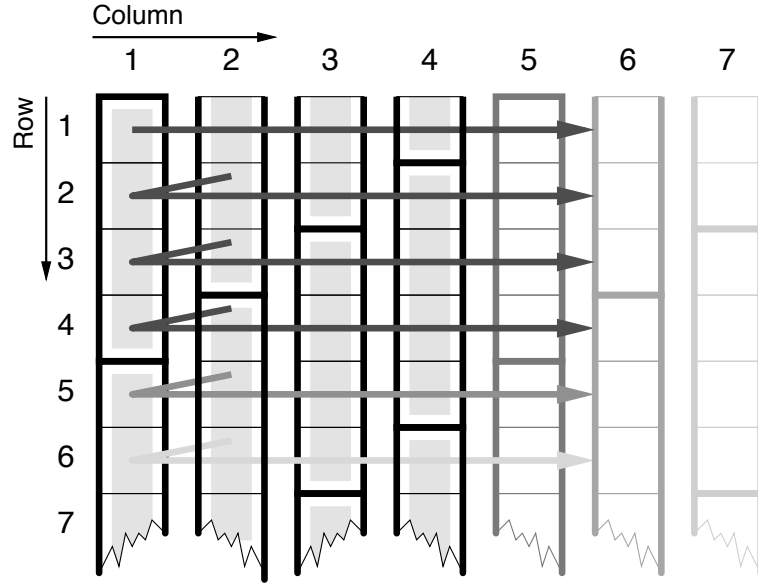
whereas 33 words are read or written for the worst case. We thus expect a 1:11 performance ratio, roughly the value observed.

We must stress again that performance predictions based on architectural specifications [M41, M44] do work in many, but not in all cases, especially on commodity systems where factors like chipsets, memory chips, interrupts, etc., are basically uncontrollable. Sometimes only a qualitative understanding of the reasons for some peculiar performance behavior can be developed, but this is often enough to derive the next logical optimization steps.

The first and most simple optimization for dense matrix transpose would consist in interchanging the order of the loop nest, i.e., pulling the  $i$  loop inside. This would render the access to matrix B strided but eliminate the strided write for A, thus saving roughly half the bandwidth (5/11, to be exact) for very large  $N$ . The measured performance gain (see inset in Figure 3.8, “flipped” graph), though noticeable, falls short of this expectation. One possible reason for this could be a slightly better efficiency of the memory interface with strided writes.

In general, the performance graphs in Figure 3.8 look quite erratic at some points. At first sight it is unclear whether some  $N$  should lead to strong performance penalties as compared to neighboring values. A closer look (“vanilla” graph in Figure 3.9) reveals that powers of two in array dimensions seem to be quite unfavorable (the benchmark program allocates new matrices with appropriate dimensions for each new  $N$ ). As mentioned in Section 1.3.2 on page 19, strided memory access leads to *thrashing* when successive iterations hit the same (set of) cache line(s) because of insufficient associativity. Figure 3.7 shows clearly that this can easily happen with matrix transpose if the leading dimension is a power of two. On a direct-mapped cache of size  $C$ , every  $C/N$ -th iteration hits the same cache line. At a line length of

**Figure 3.10:** Cache line traversal for padded matrix transpose. Padding may increase effective cache size by alleviating associativity conflicts.



$L_c$  words, the *effective cache size* is

$$C_{\text{eff}} = L_c \max \left( 1, \frac{C}{N} \right). \quad (3.11)$$

It is the number of cache words that are actually usable due to associativity constraints. On an  $m$ -way set-associative cache this number is merely multiplied by  $m$ . Considering a real-world example with  $C = 2^{17}$  (1 MByte),  $L_c = 16$ ,  $m = 8$  and  $N = 1024$  one arrives at  $C_{\text{eff}} = 2^{11}$  DP words, i.e., 16 kBytes. So  $NL_c \gg C_{\text{eff}}$  and performance should be similar to the very large  $N$  limit described above, which is roughly true.

A simple code modification, however, eliminates the thrashing effect: Assuming that matrix  $A$  has dimensions  $1024 \times 1024$ , enlarging the leading dimension by  $p$  (called *padding*) to get  $A(1024+p, 1024)$  results in a fundamentally different cache use pattern. After  $L_c/p$  iterations, the address belongs to another set of  $m$  cache lines and there is no associativity conflict if  $Cm/N > L_c/p$  (see Figure 3.10). In Figure 3.9 the striking effect of padding the leading dimension by  $p = 1$  is shown with the “padded” graph. Generally speaking, one should by all means stay away from powers of two in leading array dimensions. It is clear that different dimensions may require different paddings to get optimal results, so sometimes a rule of thumb is applied: Try to make leading array dimensions odd multiples of 16.

Further optimization approaches that can be applied to matrix transpose will be discussed in the following sections.



### 3.5 Algorithm classification and access optimizations

The optimization potential of many loops on cache-based processors can easily be estimated just by looking at basic parameters like the scaling behavior of data transfers and arithmetic operations versus problem size. It can then be decided whether investing optimization effort would make sense.

#### 3.5.1 $O(N)/O(N)$

If both the number of arithmetic operations and the number of data transfers (loads/stores) are proportional to the problem size (or “loop length”)  $N$ , optimization potential is usually very limited. Scalar products, vector additions, and sparse matrix-vector multiplication are examples for this kind of problems. They are inevitably memory-bound for large  $N$ , and compiler-generated code achieves good performance because  $O(N)/O(N)$  loops tend to be quite simple and the correct software pipelining strategy is obvious. *Loop nests*, however, are a different matter (see below).

But even if loops are not nested there is sometimes room for improvement. As an example, consider the following vector additions:

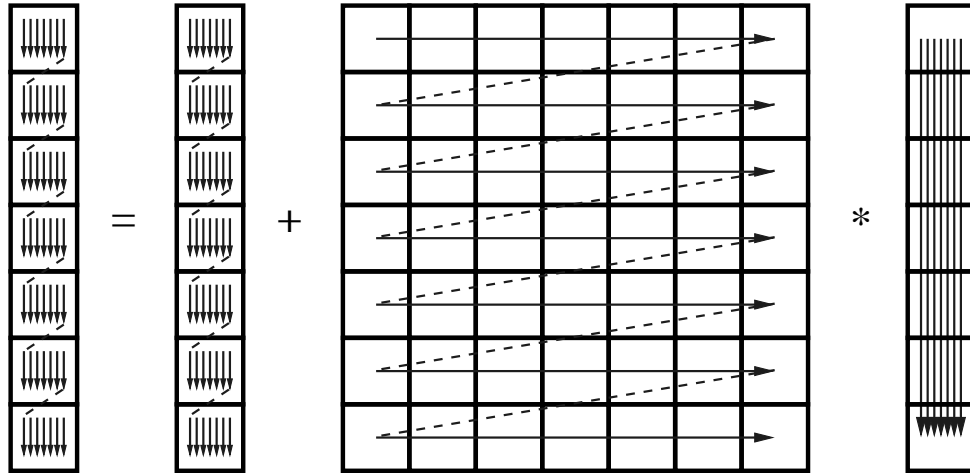
<pre> 1 do i=1,N 2   A(i) = B(i) + C(i) 3 enddo 4 do i=1,N 5   Z(i) = B(i) + E(i) 6 enddo </pre>	<p>loop fusion</p> <p>→</p>	<pre> ! optimized do i=1,N   A(i) = B(i) + C(i) ! save a load for B(i)   Z(i) = B(i) + E(i) enddo </pre>
--	-----------------------------	--

Each of the loops on the left has no options left for optimization. The code balance is 3/1 as there are two loads, one store, and one addition per loop (not counting write allocates). Array B, however, is loaded again in the second loop, which is unnecessary: *Fusing* the loops into one has the effect that each element of B only has to be loaded once, reducing code balance to 5/2. All else being equal, performance in the memory-bound case will improve by a factor of 6/5 (if write allocates cannot be avoided, this will be 8/7).

*Loop fusion* has achieved an  $O(N)$  data reuse for the two-loop constellation so that a complete load stream could be eliminated. In simple cases like the one above, compilers can often apply this optimization by themselves.

#### 3.5.2 $O(N^2)/O(N^2)$

In typical two-level loop nests where each loop has a trip count of  $N$ , there are  $O(N^2)$  operations for  $O(N^2)$  loads and stores. Examples are dense matrix-vector multiply, matrix transpose, matrix addition, etc. Although the situation on the inner level is similar to the  $O(N)/O(N)$  case and the problems are generally memory-bound, the nesting opens new opportunities. Optimization, however, is again usually limited to



**Figure 3.11:** Unoptimized  $N \times N$  dense matrix vector multiply. The RHS vector is loaded  $N$  times.

a constant factor of improvement. As an example we consider dense matrix-vector multiply (MVM):

---

```

1 do i=1,N
2   tmp = C(i)
3   do j=1,N
4     tmp = tmp + A(j,i) * B(j)
5   enddo
6   C(i) = tmp
7 enddo

```

---

This code has a balance of 1 W/F (two loads for A and B and two flops). Array C is indexed by the outer loop variable, so updates can go to a register (here clarified through the use of the scalar `tmp` although compilers can do this transformation automatically) and do not count as load or store streams. Matrix A is only loaded once, but B is loaded  $N$  times, once for each outer loop iteration (see Figure 3.11). One would like to apply the same fusion trick as above, but there are not just two but  $N$  inner loops to fuse. The solution is *loop unrolling*: The outer loop is traversed with a stride  $m$  and the inner loop is replicated  $m$  times. We thus have to deal with the situation that the outer loop count might not be a multiple of  $m$ . This case has to be handled by a remainder loop:

---

```

1 ! remainder loop
2 do r=1,mod(N,m)
3   do j=1,N
4     C(r) = C(r) + A(j,r) * B(j)
5   enddo
6 enddo
7 ! main loop
8 do i=r,N,m
9   do j=1,N
10    C(i) = C(i) + A(j,i) * B(j)
11  enddo

```

---

```

12  do j=1,N
13      C(i+1) = C(i+1) + A(j,i+1) * B(j)
14  enddo
15      ! m times
16      ...
17  do j=1,N
18      C(i+m-1) = C(i+m-1) + A(j,i+m-1) * B(j)
19  enddo
20 enddo

```

---

The remainder loop is subject to the same optimization techniques as the original loop, but otherwise unimportant. For this reason we will ignore remainder loops in the following.

By just unrolling the outer loop we have not gained anything but a considerable code bloat. However, loop fusion can now be applied easily:

---

```

1  ! remainder loop ignored
2  do i=1,N,m
3      do j=1,N
4          C(i) = C(i) + A(j,i) * B(j)
5          C(i+1) = C(i+1) + A(j,i+1) * B(j)
6          ! m times
7          ...
8          C(i+m-1) = C(i+m-1) + A(j,i+m-1) * B(j)
9      enddo
10 enddo

```

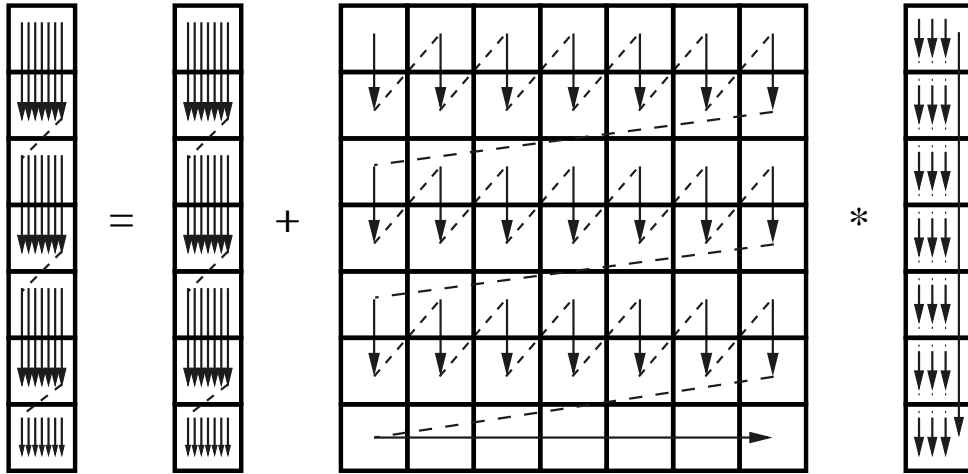
---

The combination of outer loop unrolling and fusion is often called *unroll and jam*. By  $m$ -way unroll and jam we have achieved an  $m$ -fold reuse of each element of  $B$  from register so that code balance reduces to  $(m+1)/2m$  which is clearly smaller than one for  $m > 1$ . If  $m$  is very large, the performance gain can get close to a factor of two. In this case array  $B$  is only loaded a few times or, ideally, just once from memory. As  $A$  is always loaded exactly once and has size  $N^2$ , the total memory traffic with  $m$ -way unroll and jam amounts to  $N^2(1 + 1/m) + N$ . Figure 3.12 shows the memory access pattern for two-way unrolled dense matrix-vector multiply.

All this assumes, however, that register pressure is not too large, i.e., the CPU has enough registers to hold all the required operands used inside the now quite sizeable loop body. If this is not the case, the compiler must spill register data to cache, slowing down the computation (see also Section 2.4.5). Again, compiler logs, if available, can help identify such a situation.

Unroll and jam can be carried out automatically by some compilers at high optimization levels. Be aware though that a complex loop body may obscure important information and manual optimization could be necessary, either (as shown above) by hand-coding or *compiler directives* that specify high-level transformations like unrolling. Directives, if available, are the preferred alternative as they are much easier to maintain and do not lead to visible code bloat. Regrettably, compiler directives are inherently nonportable.

The matrix transpose code from the previous section is another typical example for an  $O(N^2)/O(N^2)$  problem, although in contrast to dense MVM there is no direct



**Figure 3.12:** Two-way unrolled dense matrix vector multiply. The data traffic caused by reloading the RHS vector is reduced by roughly a factor of two. The remainder loop is only a single (outer) iteration in this example.

opportunity for saving on memory traffic; both matrices have to be read or written exactly once. Nevertheless, by using unroll and jam on the “flipped” version a significant performance boost of nearly 50% is observed (see dotted line in Figure 3.8):

---

```

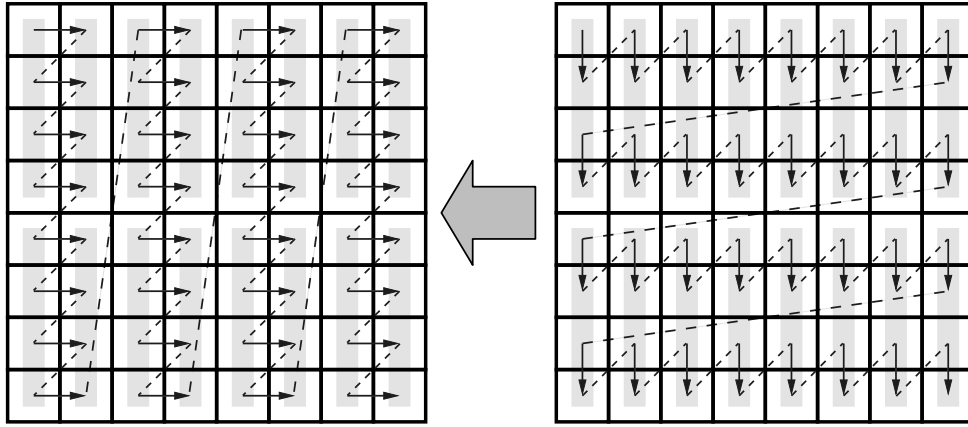
1 do j=1,N,m
2   do i=1,N
3     A(i,j)      = B(j,i)
4     A(i,j+1)    = B(j+1,i)
5     ...
6     A(i,j+m-1) = B(j+m-1,i)
7   enddo
8 enddo

```

---

Naively one would not expect any effect at  $m = 4$  because the basic analysis stays the same: In the mid- $N$  region the number of available cache lines is large enough to hold up to  $L_c$  columns of the store stream. Figure 3.13 shows the situation for  $m = 2$ . However, the fact that  $m$  words in each of the load stream’s cache lines are now accessed *in direct succession* reduces the TLB misses by a factor of  $m$ , although the TLB is still way too small to map the whole working set.

Even so, cutting down on TLB misses does not remedy the performance breakdown for large  $N$  when the cache gets too small to hold  $N$  cache lines. It would be nice to have a strategy which reuses the remaining  $L_c - m$  words of the strided stream’s cache lines right away so that each line may be evicted soon and would not have to be reclaimed later. A “brute force” method is  $L_c$ -way unrolling, but this approach leads to large-stride accesses in the store stream and is not a general solution as large unrolling factors raise register pressure in loops with arithmetic operations. *Loop blocking* can achieve optimal cache line use without additional register pressure. It does not save load or store operations but increases the cache hit ratio. For a



**Figure 3.13:** Two-way unrolled “flipped” matrix transpose (i.e., with strided load in the original version).

loop nest of depth  $d$ , blocking introduces up to  $d$  additional outer loop levels that cut the original inner loops into chunks:

---

```

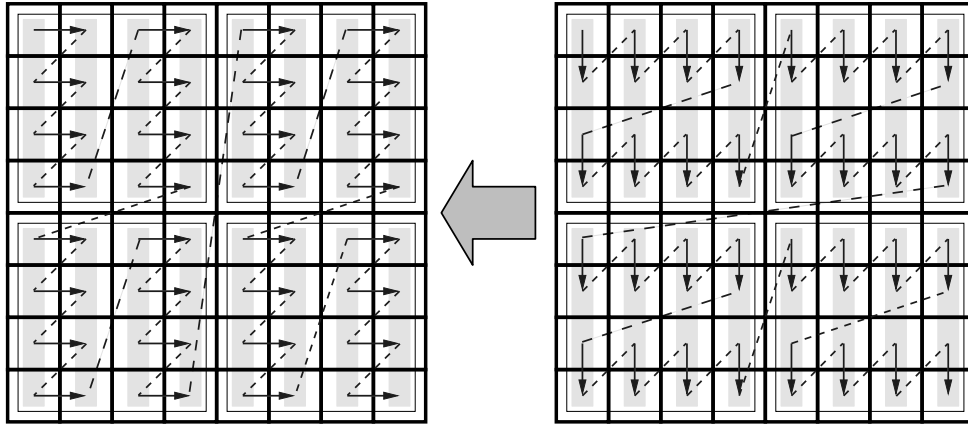
1  do jj=1,N,b
2    jstart=jj; jend=jj+b-1
3    do ii=1,N,b
4      istart=ii; iend=ii+b-1
5      do j=jstart,jend,m
6        do i=istart,iend
7          a(i,j) = b(j,i)
8          a(i,j+1) = b(j+1,i)
9          ...
10         a(i,j+m-1) = b(j+m-1,i)
11        enddo
12      enddo
13    enddo
14  enddo

```

---

In this example we have used *two-dimensional blocking* with identical blocking factors  $b$  for both loops in addition to  $m$ -way unroll and jam. This change does not alter the loop body so the number of registers needed to hold operands stays the same. However, the cache line access characteristics are much improved (see Figure 3.14 which shows a combination of two-way unrolling and  $4 \times 4$  blocking). If the blocking factors are chosen appropriately, the cache lines of the strided stream will have been used completely at the end of a block and can be evicted “soon.” Hence, we expect the large- $N$  performance breakdown to disappear. The dotted-dashed graph in Figure 3.8 demonstrates that  $50 \times 50$  blocking combined with four-way unrolling alleviates all memory access problems induced by the strided stream.

Loop blocking is a very general and powerful optimization that can often not be performed by compilers. The correct blocking factor to use should be determined experimentally through careful benchmarking, but one may be guided by typical cache sizes, i.e., when blocking for L1 cache the aggregated working set size of all blocked inner loop nests should not be much larger than half the cache. Which



**Figure 3.14:**  $4 \times 4$  blocked and two-way unrolled “flipped” matrix transpose.

cache level to block for depends on the operations performed and there is no general recommendation.

### 3.5.3 $O(N^3)/O(N^2)$

If the number of operations is larger than the number of data items by a factor that grows with problem size, we are in the very fortunate situation to have tremendous optimization potential. By the techniques described above (unroll and jam, loop blocking) it is sometimes possible for these kinds of problems to render the implementation cache-bound. Examples for algorithms that show  $O(N^3)/O(N^2)$  characteristics are dense matrix-matrix multiplication (MMM) and dense matrix diagonalization. It is beyond the scope of this book to develop a well-optimized MMM, let alone eigenvalue calculation, but we can demonstrate the basic principle by means of a simpler example which is actually of the  $O(N^2)/O(N)$  type:

---

```

1 do i=1,N
2   do j=1,N
3     sum = sum + foo(A(i),B(j))
4   enddo
5 enddo
```

---

The complete data set is  $O(N)$  here but  $O(N^2)$  operations (calls to `foo()`, additions) are performed on it. In the form shown above, array `B` is loaded from memory  $N$  times, so the total memory traffic amounts to  $N(N+1)$  words.  $m$ -way unroll and jam is possible and will immediately reduce this to  $N(N/m+1)$ , but the disadvantages of large unroll factors have been pointed out already. Blocking the inner loop with a blocksize of  $b$ , however,

---

```

1 do jj=1,N,b
2   jstart=jj; jend=jj+b-1
3   do i=1,N
4     do j=jstart,jend
5       sum = sum + foo(A(i),B(j))
```

---

```

6      enddo
7  enddo
8  enddo

```

---

has two effects:

- Array B is now loaded only once from memory, provided that  $b$  is small enough so that  $b$  elements fit into cache and stay there as long as they are needed.
- Array A is loaded from memory  $N/b$  times instead of once.

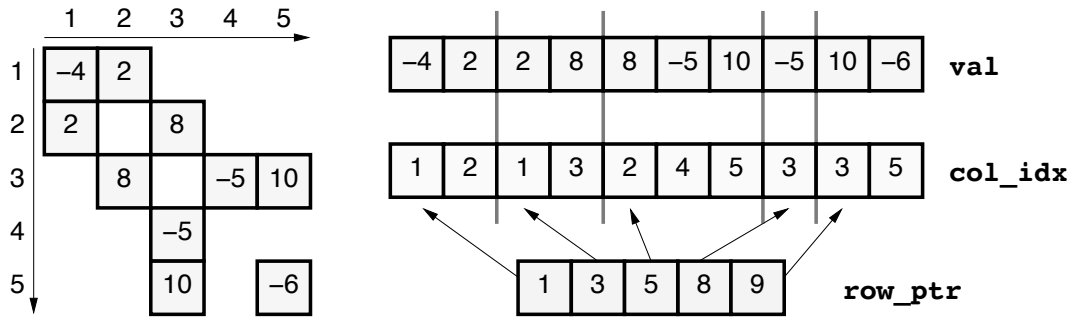
Although A is streamed through cache  $N/b$  times, the probability that the current block of B will be evicted is quite low, the reason being that those cache lines are used very frequently and thus kept by the LRU replacement algorithm. This leads to an effective memory traffic of  $N(N/b + 1)$  words. As  $b$  can be made much larger than typical unrolling factors, blocking is the best optimization strategy here. Unroll and jam can still be applied to enhance in-cache code balance. The basic  $N^2$  dependence is still there, but with a prefactor that can make the difference between memory-bound and cache-bound behavior. A code is cache-bound if main memory bandwidth and latency are not the limiting factors for performance any more. Whether this goal is achievable on a certain architecture depends on the cache size, cache and memory speeds, and the algorithm, of course.

Algorithms of the  $O(N^3)/O(N^2)$  type are typical candidates for optimizations that can potentially lead to performance numbers close to the theoretical maximum. If blocking and unrolling factors are chosen appropriately, dense matrix-matrix multiply, e.g., is an operation that usually achieves over 90% of peak for  $N \times N$  matrices if  $N$  is not too small. It is provided in highly optimized versions by system vendors as, e.g., contained in the BLAS (Basic Linear Algebra Subsystem) library. One might ask why unrolling should be applied at all when blocking already achieves the most important task of making the code cache-bound. The reason is that even if all the data resides in a cache, many processor architectures do not have the capability for sustaining enough loads and stores per cycle to feed the arithmetic units continuously. For instance, the current x86 processors from Intel can sustain one load and one store operation per cycle, which makes unroll and jam mandatory if the kernel of a loop nest uses more than one load stream, especially in cache-bound situations like the blocked  $O(N^2)/O(N)$  example above.

Although demonstrated here for educational purposes, there is no need to hand-code and optimize standard linear algebra and matrix operations. They should always be used from optimized libraries, if available. Nevertheless, the techniques described can be applied in many real-world codes. An interesting example with some complications is sparse matrix-vector multiply (see Section 3.6).







**Figure 3.16:** CRS sparse matrix storage format.

CRS is well-suited for cache-based microprocessors while JDS supports dependency and loop structures that are favorable on vector systems.

In CRS, an array `val` of length  $N_{nz}$  is used to store all nonzeros of the matrix, row by row, without any gaps, so some information about which element of `val` originally belonged to which row and column must be supplied. This is done by two additional integer arrays, `col_idx` of length  $N_{nz}$  and `row_ptr` of length  $N_r$ . `col_idx` stores the column index of each nonzero in `val`. `row_ptr` contains the indices at which new rows start in `val` (see Figure 3.16). The basic code to perform an MVM using this format is quite simple:

---

```

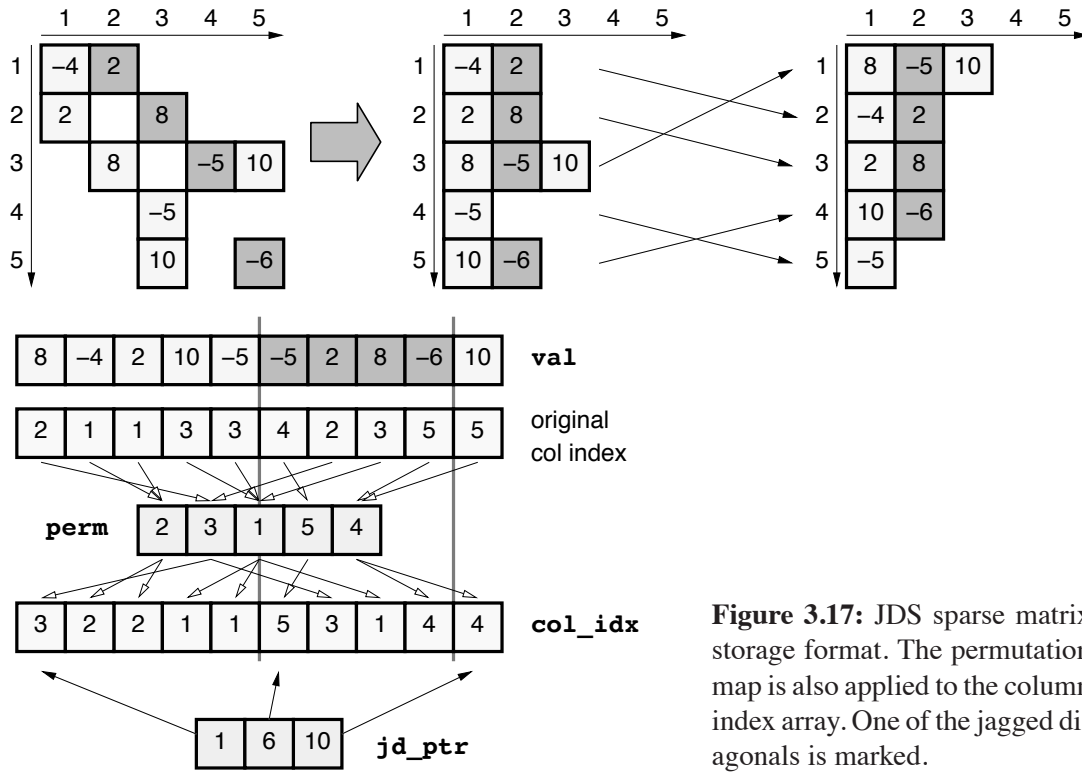
1 do i = 1, Nr
2   do j = row_ptr(i), row_ptr(i+1) - 1
3     C(i) = C(i) + val(j) * B(col_idx(j))
4   enddo
5 enddo

```

---

The following points should be noted:

- There is a long outer loop (length  $N_r$ ).
- The inner loop may be “short” compared to typical microprocessor pipeline lengths.
- Access to result vector `C` is well optimized: It is only loaded once from main memory.
- The nonzeros in `val` are accessed with stride one.
- As expected, the RHS vector `B` is accessed indirectly. This may, however, not be a serious performance problem depending on the exact structure of the matrix. If the nonzeros are concentrated mainly around the diagonal, there will even be considerable spatial and/or temporal locality.
- $B_c = 5/4$  W/F if the integer load to `col_idx` is counted with four bytes. We are neglecting the possibly much larger transfer volume due to partially used cache lines.



**Figure 3.17:** JDS sparse matrix storage format. The permutation map is also applied to the column index array. One of the jagged diagonals is marked.

Some of those points will be of importance later when we demonstrate parallel sMVM (see Section 7.3 on page 181).

JDS requires some rearrangement of the matrix entries beyond simple zero elimination. First, all zeros are eliminated from the matrix rows and the nonzeros are shifted to the left. Then the matrix rows are sorted by descending number of nonzeros so that the longest row is at the top and the shortest row is at the bottom. The permutation map generated during the sorting stage is stored in array **perm** of length  $N_r$ . Finally, the now established columns are stored in array **val** consecutively. These columns are also called *jagged diagonals* as they traverse the original sparse matrix from left top to right bottom (see Figure 3.17). For each nonzero the original column index is stored in **col\_idx** just like in the CRS. In order to have the same element order on the RHS and LHS vectors, the **col\_idx** array is subject to the above-mentioned permutation as well. Array **jd\_ptr** holds the start indices of the  $N_j$  jagged diagonals. A standard code for sMVM in JDS format is only slightly more complex than with CRS:

---

```

1 do diag=1, Nj
2   diagLen = jd_ptr(diag+1) - jd_ptr(diag)
3   offset = jd_ptr(diag) - 1
4   do i=1, diagLen
5     C(i) = C(i) + val(offset+i) * B(col_idx(offset+i))
6   enddo
7 enddo

```

---

The `perm` array storing the permutation map is not required here; usually, all sMVM operations are done in permuted space. These are the notable properties of this loop:

- There is a long inner loop without dependencies, which makes JDS a much better storage format for vector processors than CRS.
- The outer loop is short (number of jagged diagonals).
- The result vector is loaded multiple times (at least partially) from memory, so there might be some optimization potential.
- The nonzeros in `val` are accessed with stride one.
- The RHS vector is accessed indirectly, just as with CRS. The same comments as above do apply, although a favorable matrix layout would feature straight diagonals, not compact rows. As an additional complication the matrix rows as well as the RHS vector are permuted.
- $B_c = 9/4$  W/F if the integer load to `col_idx` is counted with four bytes.

The code balance numbers of CRS and JDS sMVM seem to be quite in favor of CRS.

### 3.6.2 Optimizing JDS sparse MVM

Unroll and jam should be applied to the JDS sMVM, but it usually requires the length of the inner loop to be independent of the outer loop index. Unfortunately, the jagged diagonals are generally not all of the same length, violating this condition. However, an optimization technique called *loop peeling* can be employed which, for  $m$ -way unrolling, cuts rectangular  $m \times x$  chunks and leaves  $m - 1$  partial diagonals over for separate treatment (see Figure 3.18; the remainder loop is omitted as usual):

---

```

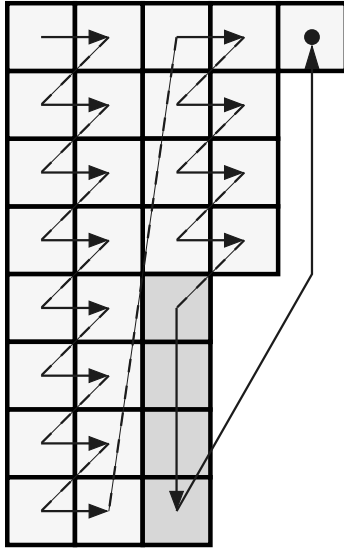
1  do diag=1,Nj,2  ! two-way unroll & jam
2    diagLen = min( (jd_ptr(diag+1)-jd_ptr(diag)) , \
3                  (jd_ptr(diag+2)-jd_ptr(diag+1)) )
4    offset1 = jd_ptr(diag) - 1
5    offset2 = jd_ptr(diag+1) - 1
6    do i=1, diagLen
7      C(i) = C(i)+val(offset1+i)*B(col_idx(offset1+i))
8      C(i) = C(i)+val(offset2+i)*B(col_idx(offset2+i))
9    enddo
10   ! peeled-off iterations
11   offset1 = jd_ptr(diag)
12   do i=(diagLen+1),(jd_ptr(diag+1)-jd_ptr(diag))
13     c(i) = c(i)+val(offset1+i)*b(col_idx(offset1+i))
14   enddo
15 enddo

```

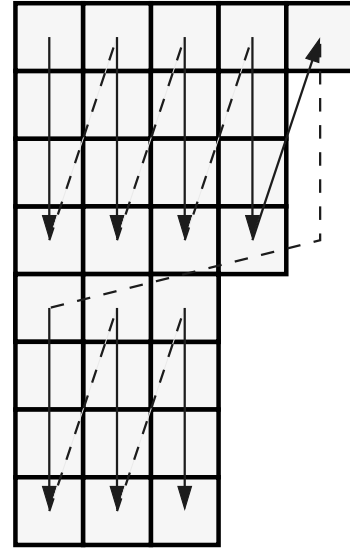
---

Assuming that the peeled-off iterations account for a negligible contribution to CPU time,  $m$ -way unroll and jam reduces code balance to

$$B_c = \left( \frac{1}{m} + \frac{5}{4} \right) \text{W/F}.$$



**Figure 3.18:** JDS matrix traversal with two-way unroll and jam and loop peeling. The peeled iterations are marked.



**Figure 3.19:** JDS matrix traversal with four-way loop blocking.

If  $m$  is large enough, this can get close to the CRS balance. However, as explained before large  $m$  leads to strong register pressure and is not always desirable. Generally, a sensible combination of unrolling and blocking is employed to reduce memory traffic and enhance in-cache performance at the same time. Blocking is indeed possible for JDS sMVM as well (see Figure 3.19):

---

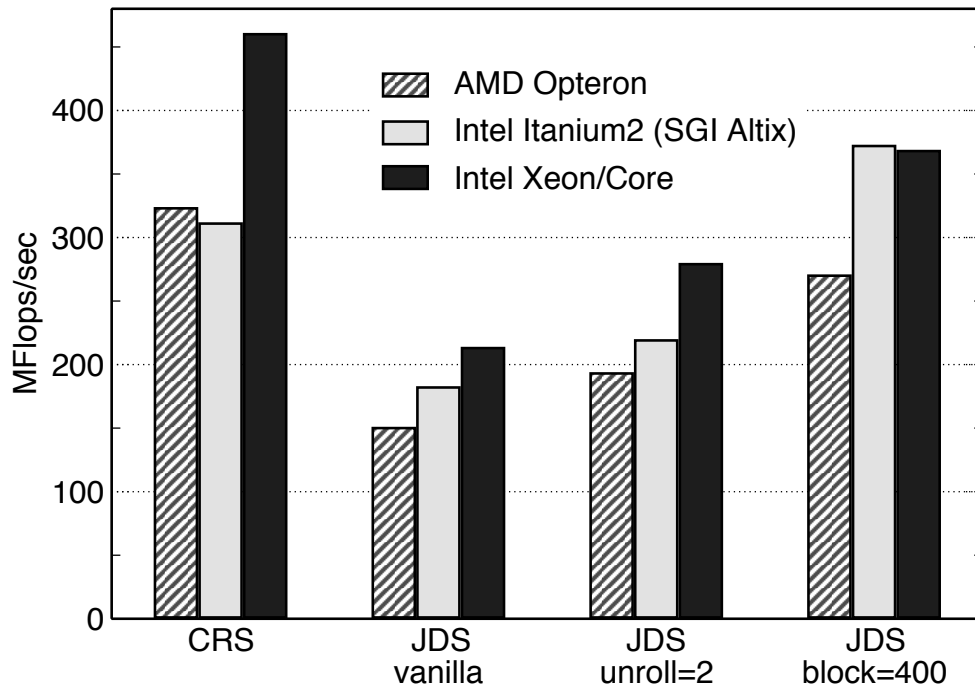
```

1  ! loop over blocks
2  do ib=1, Nr, b
3    block_start = ib
4    block_end   = min(ib+b-1, Nr)
5    ! loop over diagonals in one block
6    do diag=1, Nj
7      diagLen = jd_ptr(diag+1)-jd_ptr(diag)
8      offset = jd_ptr(diag) - 1
9      if(diagLen .ge. block_start) then
10         ! standard JDS sMVM kernel
11         do i=block_start, min(block_end,diagLen)
12           B(i) = B(i)+val(offset+i)*B(col_idx(offset+i))
13         enddo
14       endif
15     enddo
16  enddo

```

---

With this optimization the result vector is effectively loaded only once from memory if the block size  $b$  is not too large. The code should thus get similar performance as the CRS version, although code balance has not been changed. As anticipated above with dense matrix transpose, blocking does not optimize for register reuse but for cache utilization.



**Figure 3.20:** Performance comparison of sparse MVM codes with different optimizations. A matrix with  $1.7 \times 10^7$  unknowns and 20 jagged diagonals was chosen. The blocking size of 400 has proven to be optimal for a wide range of architectures.

Figure 3.20 shows a performance comparison of CRS and plain, two-way unrolled and blocked ( $b = 400$ ) JDS sMVM on three different architectures, for a test matrix from solid state physics (six-site one-dimensional Holstein-Hubbard model at half filling). The CRS variant seems to be preferable for standard AMD and Intel microprocessors, which is not surprising because it features the lowest code balance right away without any subsequent manual optimizations and the short inner loop length is less unfavorable on CPUs with out-of-order capabilities. The Intel Itanium2 processor with its EPIC architecture [V113], however, shows mediocre performance for CRS and tops at the blocked JDS version. This architecture cannot cope very well with the short loops of CRS due to the absence of out-of-order processing and the compiler, despite detecting all instruction-level parallelism on the inner loop level, not being able to overlap the wind-down of one row with the wind-up phase of the next. This effect would certainly be much more pronounced if the working set did fit into the cache [O56].

## Problems

For solutions see page 289 ff.

3.1 *Strided access*. How do the balance and lightspeed considerations in Sec-

tion 3.1 have to be modified if one or more arrays are accessed with nonunit stride? What kind of performance characteristic do you expect for a stride- $s$  vector triad,

---

```

1 do i=1,N,s
2   A(i) = B(i) + C(i) * D(i)
3 enddo

```

---

with respect to  $s$  if  $N$  is large?

3.2 *Balance fun.* Calculate code balance for the following loop kernels, assuming that all arrays have to be loaded from memory and ignoring the latency problem (appropriate loops over the counter variables  $i$  and  $j$  are always implied):

- (a)  $Y(j) = Y(j) + A(i, j) * B(i)$  (matrix-vector multiply)
- (b)  $s = s + A(i) * A(i)$  (vector norm)
- (c)  $s = s + A(i) * B(i)$  (scalar product)
- (d)  $s = s + A(i) * B(K(i))$  (scalar product with indirect access)

All arrays are of DP floating-point type except  $K$  which stores 4-byte integers.  $s$  is a double precision scalar. Calculate expected application performance based on theoretical peak bandwidth and STREAM bandwidths in MFlops/sec for those kernels on one core of a Xeon 5160 processor and on the prototypical vector processor described in Section 1.6. The Xeon CPU has a cache line size of 64 bytes. You may assume that  $N$  is large so that the arrays do not fit into any cache. For case (d), give numbers for best and worst case scenarios on the Xeon.

3.3 *Performance projection.* In future mainstream microarchitectures, SIMD capabilities will be greatly enhanced. One of the possible new features is that x86 processors will be capable of executing MULT and ADD instructions on 256-bit (instead of 128-bit) registers, i.e., four DP floating-point values, concurrently. This will effectively double the peak performance per cycle from 4 to 8 flops, given that the L1 cache bandwidth is improved by the same factor. Assuming that other parameters like memory bandwidth and clock speed stay the same, estimate the expected performance gain for using this feature, compared to a single current Intel “Core i7” core (effective STREAM-based machine balance of 0.12 W/F). Assume a perfectly SIMD-vectorized application that today spends 60% of its compute time on code that has a balance of 0.04 W/F and the remaining 40% in code with a balance of 0.5 W/F. If the manufacturers chose to extend SIMD capabilities even more, e.g., by introducing very large vector lengths, what is the absolute limit for the expected performance gain in this situation?

3.4 *Optimizing 3D Jacobi.* Generalize the 2D Jacobi algorithm introduced in Section 3.3 to three dimensions. Do you expect a change in performance char-

acteristics with varying inner loop length (Figure 3.6)? Considering the optimizations for dense matrix transpose (Section 3.4), can you think of a way to eliminate some of the performance breakdowns?

3.5 *Inner loop unrolling revisited.* Up to now we have encountered the possibility of unrolling *inner* loops only in the contexts of software pipelining and SIMD optimizations (see Chapter 2). Can inner loop unrolling also improve code balance in some situations? What are the prospects for improving the performance of a Jacobi solver by unrolling the inner loop?

3.6 *Not unrollable?* Consider the multiplication of a lower triangular matrix with a vector:

---

```

1  do r=1,N
2    do c=1,r
3      y(r) = y(r) + a(c,r) * x(c)
4    enddo
5  enddo

```

---

Can you apply “unroll and jam” to the outer loop here (see Section 3.5.2 on page 81) to reduce code balance? Write a four-way unrolled version of above code. No special assumptions about  $N$  may be made (other than being positive), and no matrix elements of  $A$  may be accessed that are outside the lower triangle (including the diagonal).

3.7 *Application optimization.* Which optimization strategies would you suggest for the piece of code below? Write down a transformed version of the code which you expect to give the best performance.

---

```

1  double precision, dimension(N,N) :: mat,s
2  double precision :: val
3  integer :: i,j
4  integer, dimension(N) :: v
5  ! ... v and s may be assumed to hold valid data
6  do i=1,N
7    do j=1,N
8      val = DBLE(MOD(v(i),256))
9      mat(i,j) = s(i,j)*(SIN(val)*SIN(val)-COS(val)*COS(val))
10    enddo
11  enddo

```

---

No assumptions about the size of  $N$  may be made. You may, however, assume that the code is part of a subroutine which gets called very frequently.  $s$  and  $v$  may change between calls, and all elements of  $v$  are positive.

3.8 *TLB impact.* The translation lookaside buffer (TLB) of even the most modern processors is scarcely large enough to even store the mappings of all memory pages that reside in the outer-level data cache. Why are TLBs so small? Isn’t this a performance bottleneck by design? What are the benefits of larger pages?

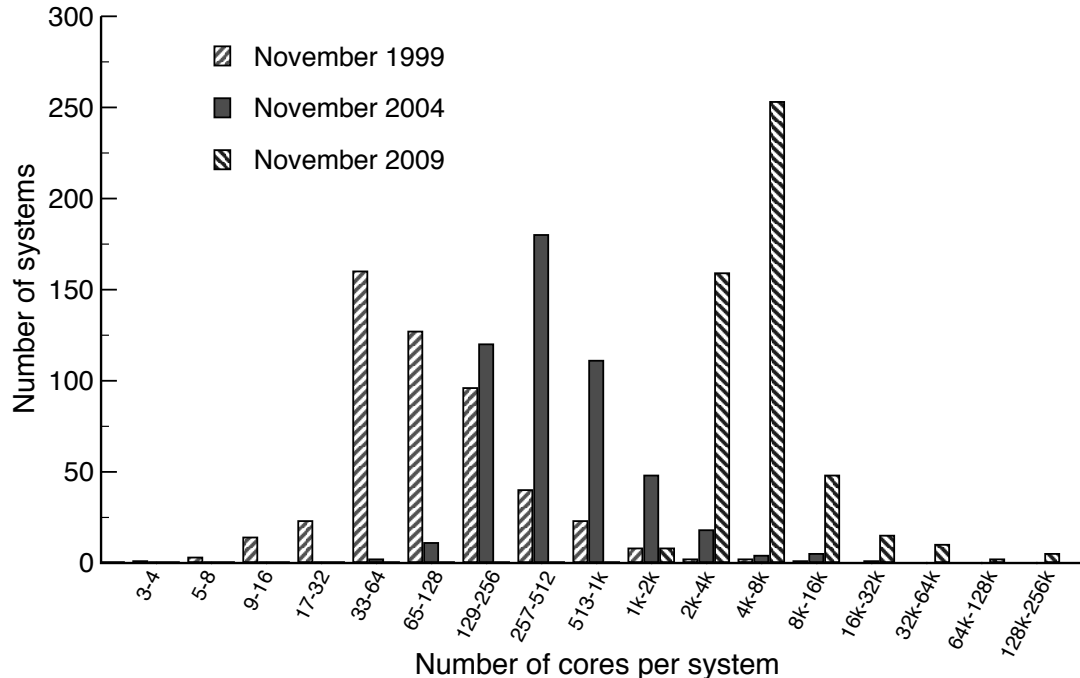




# Chapter 4

## 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:

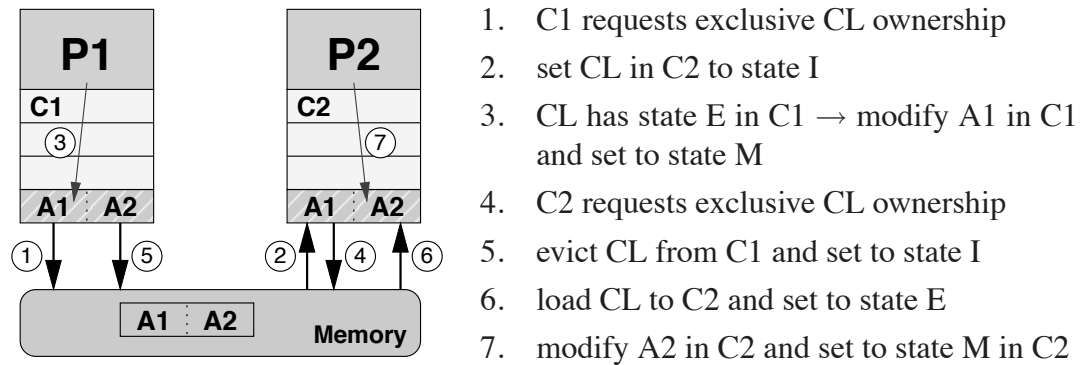
- *Uniform Memory Access* (UMA) systems exhibit a “flat” memory model: Latency and bandwidth are the same for all processors and all memory locations. This is also called *symmetric multiprocessing* (SMP). At the time of writing, single multicore processor chips (see Section 1.4) are “UMA machines.” 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



**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.

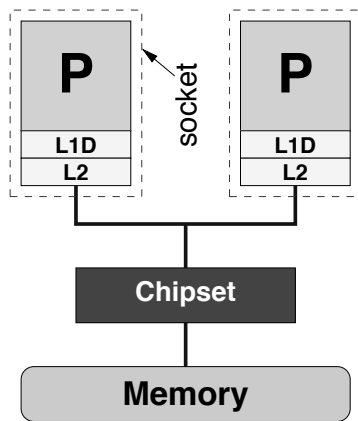
**E** *exclusive*: The cache line has been read from memory but not (yet) modified. However, it resides in no other cache.

**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.

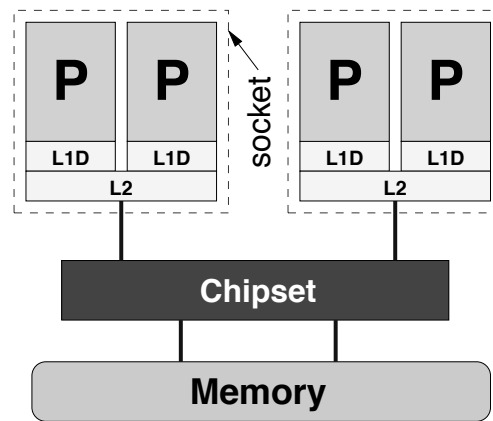
**I** *invalid*: The cache line does not reflect any sensible data. Under normal circumstances this happens if the cache line was in the shared state and another processor has requested exclusive ownership.

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.

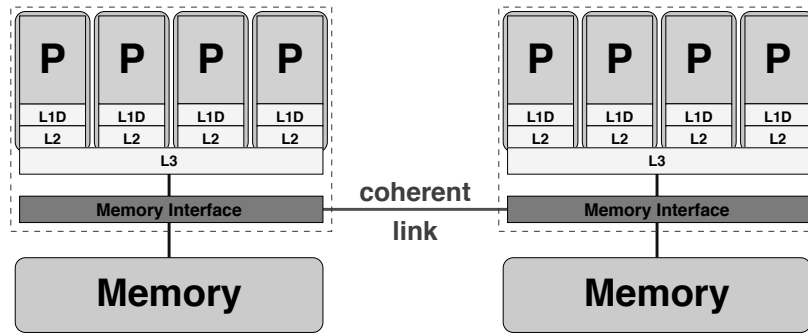
## 4.2.2 UMA

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).

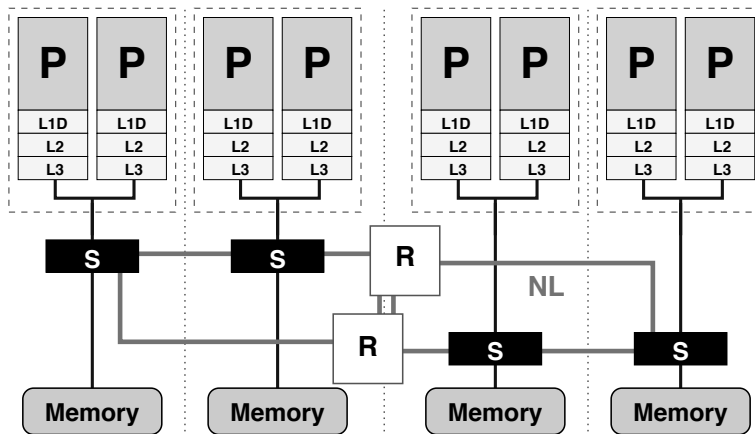
In order to maintain scalability of memory bandwidth with CPU number, non-blocking *crossbar switches* can be built that establish point-to-point connections between sockets and memory modules, similar to the chipset in Figure 4.4. Due to the very large aggregated bandwidths those become very expensive 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





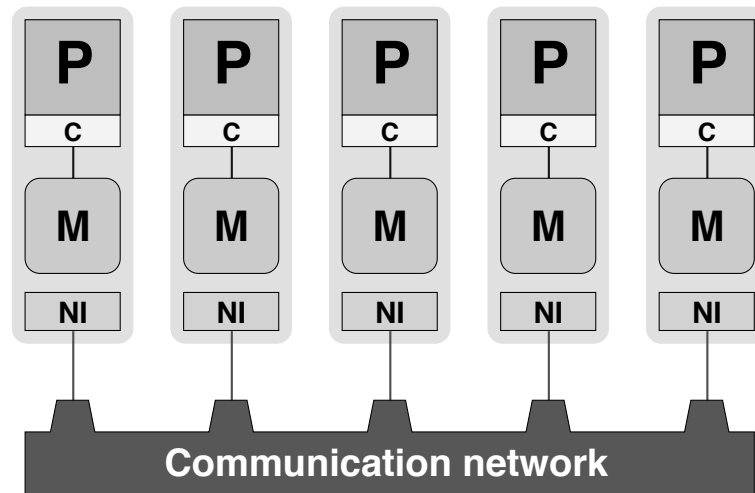
**Figure 4.6:** A ccNUMA system (SGI Altix) with four locality domains, each comprising one socket with two cores. The LDs are connected via a routed NUMALink (NL) network using routers (R).

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.



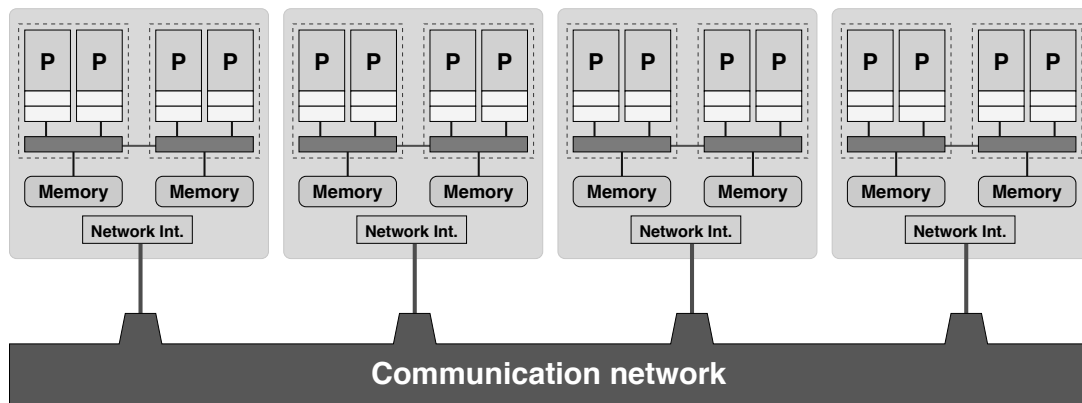
between compute nodes. If data arrives at the “wrong” locality domain, written by an I/O driver that has positioned its buffer space disregarding any ccNUMA constraints, it should be copied to its optimal destination, reducing effective bandwidth by a factor of four (three if write allocates can be avoided, 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.

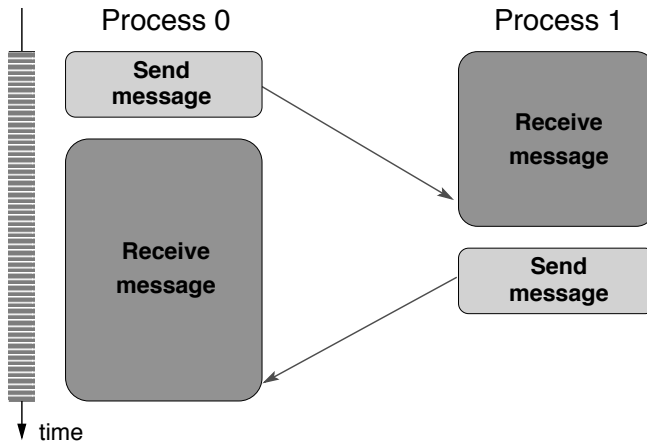
Parallel computers with hierarchical structures as described above are also called *hybrids*. The concept is actually more generic and can also be used to categorize any system with a mixture of available programming paradigms on different hardware layers. Prominent examples are clusters built from nodes 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,

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

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

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

Note that in the most general case  $T_\ell$  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:

---

```

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 microsecs
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

```

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].

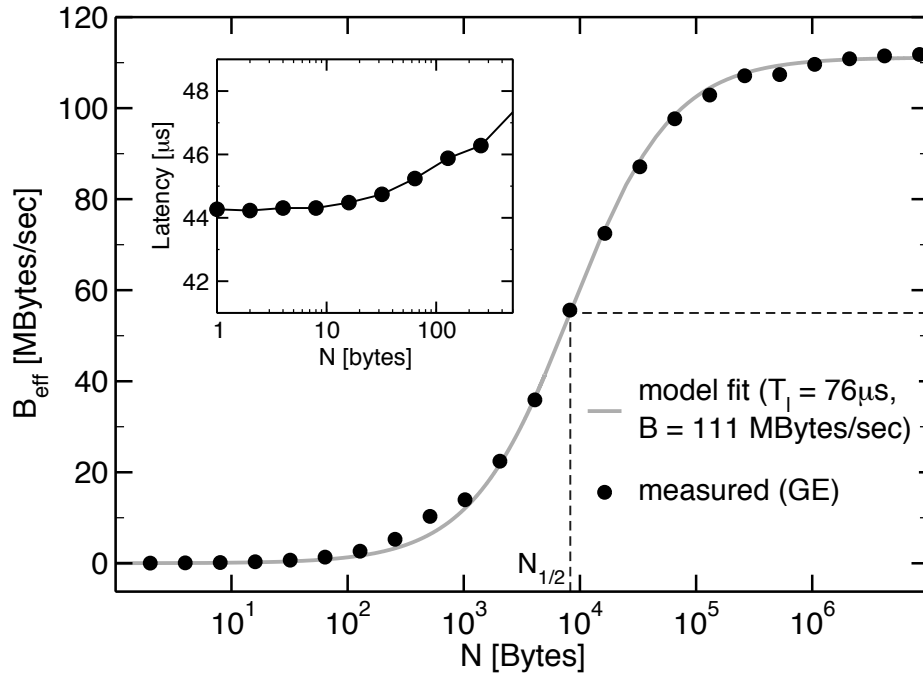
In Figure 4.10, the model parameters in (4.2) are fitted to real data measured on a Gigabit Ethernet network. This simple model is able to describe the gross features well: We observe very low bandwidth for small message sizes, because latency dominates the transfer time. For very large messages, latency 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  $T_\ell$  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:

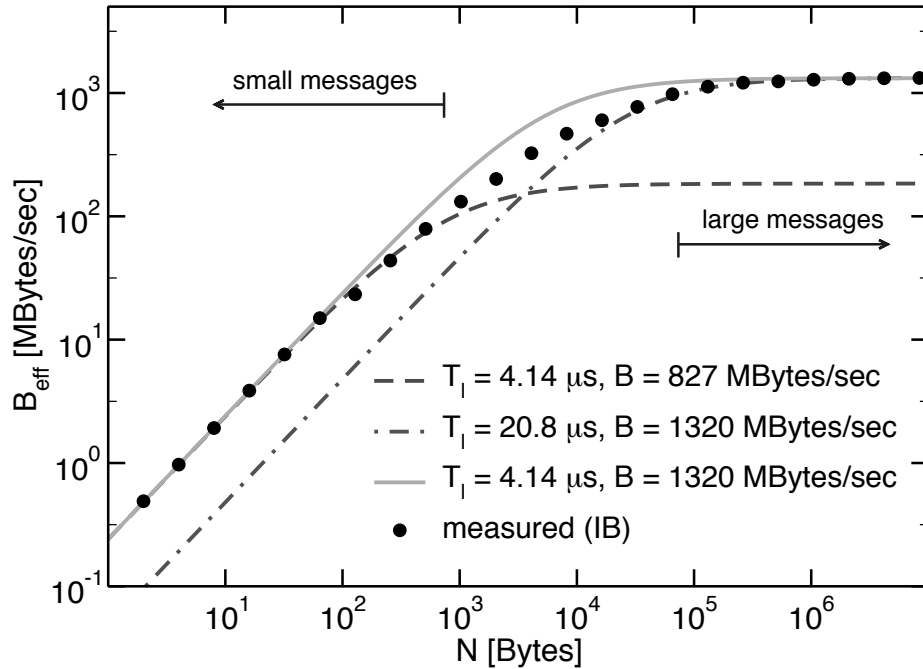
- All data transmission protocols have some overhead in the form of administrative data like message headers, etc.
- Some protocols (like, e.g., TCP/IP as used over Ethernet) define minimum message sizes, so even if the application sends a single byte, a small “frame” of  $N > 1$  bytes is transmitted.
- 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  $T_\ell$  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  $T_\ell$  (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  $T_\ell$ . 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.

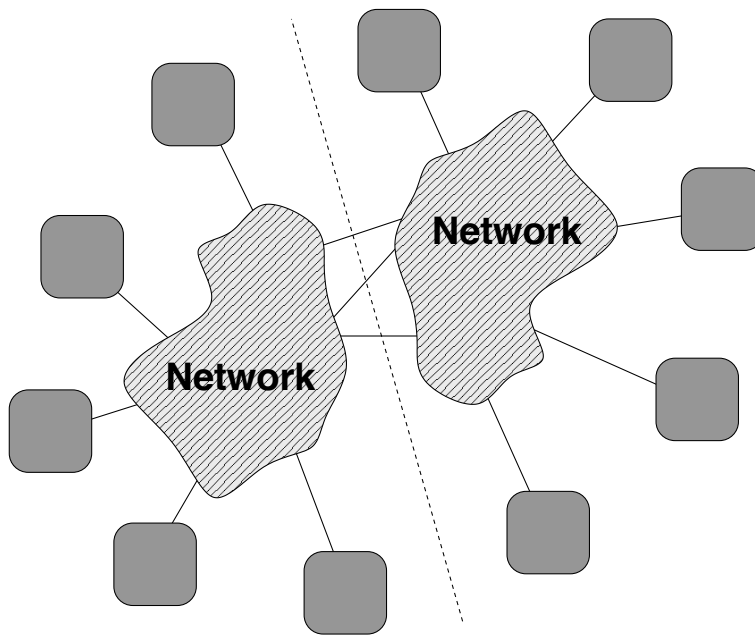
Although the saturation bandwidth  $B$  can be quite high (there are systems where the achievable internode network bandwidth is comparable to the local memory bandwidth of the processor), many applications work in a region on the bandwidth graph where latency effects still play a dominant role. To quantify 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  $\beta$  is really beneficial for all messages. At message size  $N$ , the improvement in effective bandwidth is

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

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  $\beta$ , 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.

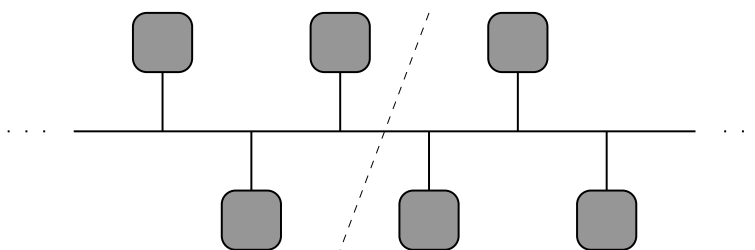


**Figure 4.12:** The bisection bandwidth  $B_b$  is the sum of the bandwidths of the minimal number of connections cut (three in this example) when dividing the system into two equal parts.

### 4.5.2 Buses

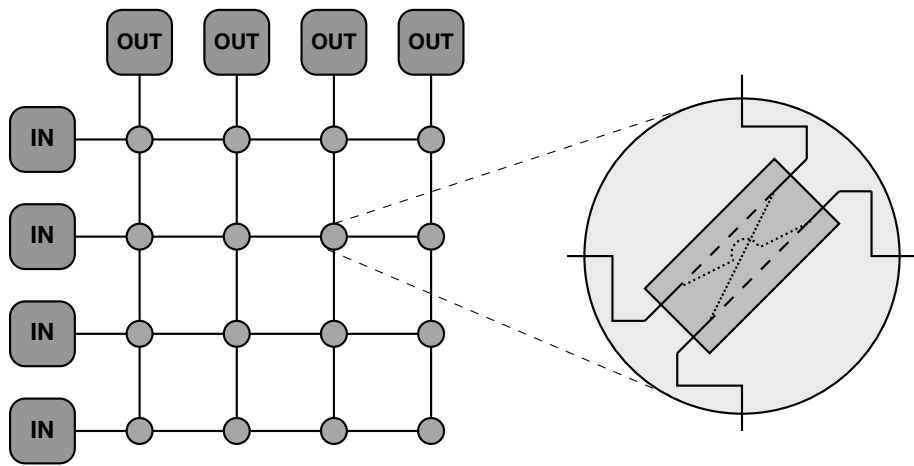
A bus is a shared medium that can be used by exactly one communicating device at a time (Figure 4.13). Some appropriate hardware mechanism must be present that detects collisions (i.e., attempts by two or more devices to transmit concurrently). Buses are very common in computer systems. They are easy 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  $2 \times 2$  switching element. The whole circuit can act as a four-port nonblocking switch.

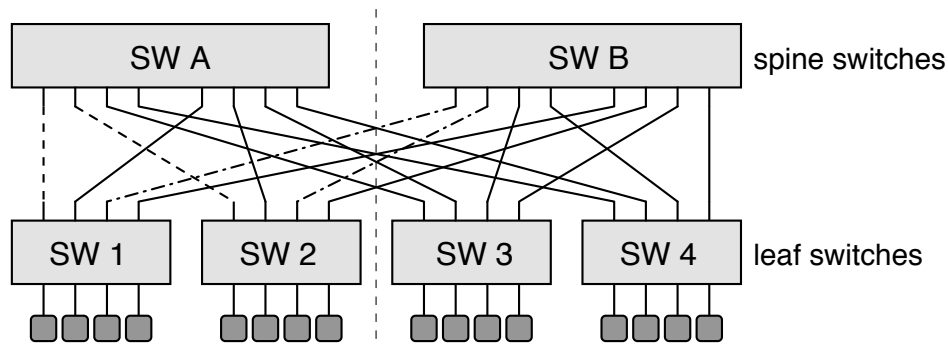
### 4.5.3 Switched and fat-tree networks

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-

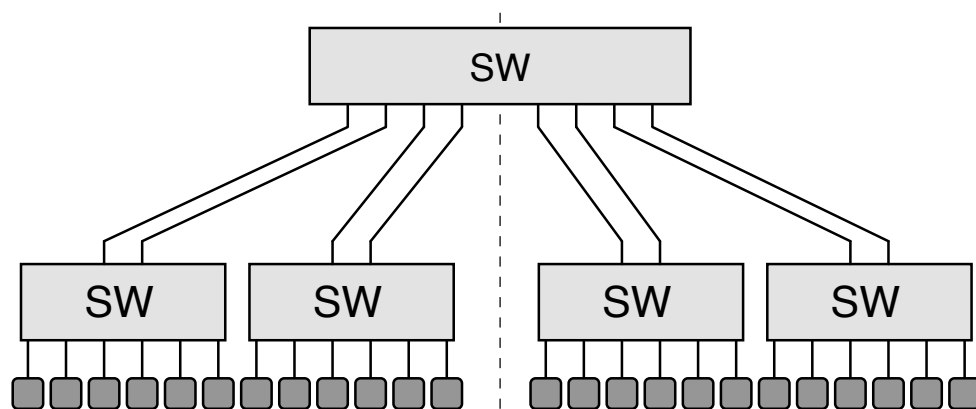




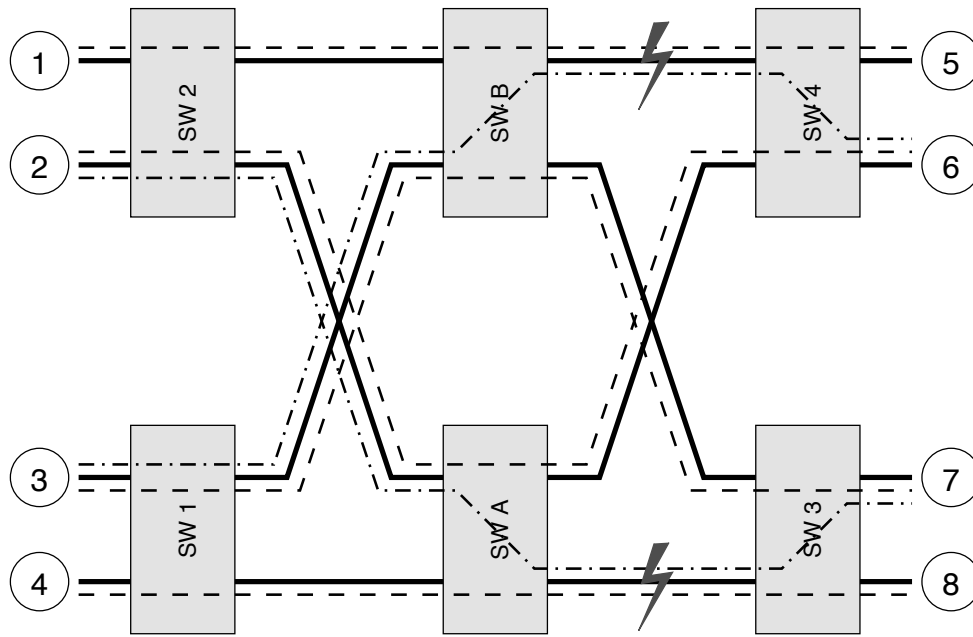
**Figure 4.15:** A fully nonblocking full-bandwidth fat-tree network with two switch layers. The switches connected to the actual compute elements are called *leaf switches*, whereas the upper layers form the *spines* of the hierarchy.

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.

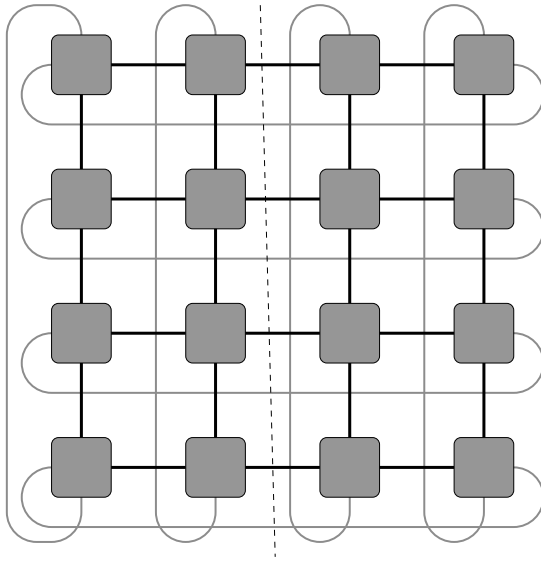


**Figure 4.17:** Even in a fully nonblocking fat-tree switch hierarchy (network cabling shown as solid lines), not all possible combinations of  $N/2$  point-to-point connections allow collision-free operation under static routing. When, starting from the collision-free connection pattern shown with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

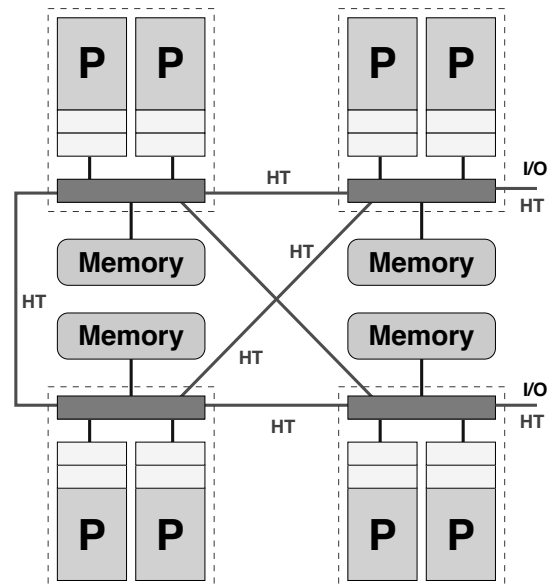
Certainly, bisection bandwidth does not scale linearly when enlarging the system in all dimensions but behaves like  $B_b(N) \propto N^{(d-1)/d}$  ( $d$  being the number of dimensions), which leads to  $B_b(N)/N \rightarrow 0$  for large  $N$ . Maximum latency scales like  $N^{1/d}$ . Although these 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?