27 Optimization Techniques for Modern High Performance Computers

Georg Hager and Gerhard Wellein

Regionales Rechenzentrum Erlangen der Friedrich-Alexander-Universität
Erlangen-Nürnberg, 91058 Erlangen, Germany

The rapid development of faster and more capable processors and architectures has often led to the false conclusion that the next generation of hardware will easily meet the scientist's requirements. This view is at fault for two reasons: First, utilizing the full power of existing systems by proper parallelization and optimization strategies, one can gain a competitive advantage without waiting for new hardware. Second, computer industry has now reached a turning point where exponential growth of compute power has ended and single-processor performance will stagnate at least for the next couple of years. The advent of multi-core CPUs was triggered by this development, making the need for more advanced, parallel, and well-optimized algorithms imminent.

This chapter describes different ways to write efficient code on current supercomputer systems. In Sect. 27.1, simple common sense optimizations for scalar code like strength reduction, correct layout of data structures and tabulation are covered first. Many scientific programs are limited by the speed of the computer system's memory interface, so it is vital to avoid slow data paths or, if this is not possible, at least use them efficiently. After some theoretical considerations on data access and performance estimates based on code analysis and hardware characteristics, techniques like loop transformations and cache blocking are explained using examples from linear algebra (matrix-vector multiplication, matrix transpose). The importance of interpreting compiler logs is emphasized. Along the discussion of performance measurements for vanilla and optimized codes we introduce peculiarities like cache thrashing and translation look-aside buffer misses, both potential showstoppers for compute performance. In a case study we apply the acquired knowledge on sparse matrix-vector multiplication, a performance-determining operation required for practically all sparse diagonalization algorithms.

Turning to shared-memory parallel programming in Sect. 27.2, we identify typical pitfalls (OpenMP loop overhead and false sharing) that can severely limit parallel scalability, and show some ways to circumvent them. The abundance of AMD Opteron nodes in clusters has initiated the necessity for optimizing memory locality. ccNUMA can lead to diverse bandwidth bottlenecks, and few compilers support special features for ensuring memory locality. Programming techniques which can alleviate ccNUMA effects are therefore described in detail using a parallelized sparse matrix-vector multiplication as a nontrivial but instructive example.

G. Hager and G. Wellein: *Optimization Techniques for Modern High Performance Computers*, Lect. Notes Phys. **739**, 731–767 (2008)
DOI 10.1007/978-3-540-74686-7_27

27.1 Optimizing Serial Code

In the age of multi-1000-processor parallel computers, writing code that runs efficiently on a single CPU has grown slightly old-fashioned in some circles. The argument for this point of view is derived from the notion that it is easier to add more CPUs and boasting massive parallelism instead of investing effort into serial optimization.

Nevertheless there can be no doubt that single-processor optimizations are of premier importance. If a speedup of two can be gained by some straightforward common sense optimization as described in the following section, the user will be satisfied with half the number of CPUs in the parallel case. In the face of Amdahl's law the benefit will usually be even larger. This frees resources for other users and projects and puts the hardware that was often acquired for considerable amounts of money to better use. If an existing parallel code is to be optimized for speed, it must be the first goal to make the single-processor run as fast as possible.

27.1.1 Common Sense Optimizations

Often very simple changes to code can lead to a significant performance boost. The most important common sense guidelines regarding the avoidance of performance pitfalls are summarized in the following. Those may seem trivial, but experience shows that many scientific codes can be improved by the simplest of measures.

27.1.1.1 Do Less Work!

In all but the rarest of cases, rearranging the code such that less work than before is being done will improve performance. A very common example is a loop that checks a number of objects to have a certain property, but all that matters in the end is that *any* object has the property at all:

```
logical FLAG
FLAG = .false.
do i=1,N
  if(complex_func(A(i)) < THRESHOLD) then
    FLAG = .true.
  endif
enddo
```

If `complex_func()` has no side effects, the only information that gets communicated to the outside of the loop is the value of `FLAG`. In this case, depending on the probability for the conditional to be true, much computational effort can be saved by leaving the loop as soon as `FLAG` changes state:

```
logical FLAG
FLAG = .false.
do i=1,N
  if(complex_func(A(i)) < THRESHOLD) then
    FLAG = .true.
    exit
  endif
enddo
```

27.1.1.2 Avoid Expensive Operations!

Sometimes, implementing an algorithm is done in a thoroughly one-to-one way, translating formulae to code without any reference to performance issues. While this is actually good (performance optimization always bears the slight danger of changing numerics, if not results), in a second step all those operations should be eliminated that can be substituted by cheaper alternatives. Prominent examples for such strong operations are trigonometric functions or exponentiation. Bear in mind that an expression like `x**2.0` is often not optimized by the compiler to become `x*x` but left as it stands, resulting in the evaluation of an exponential and a logarithm. The corresponding optimization is called strength reduction. Apart from the simple case described above, strong operations often appear with a limited set of fixed arguments. This is an example from a simulation code for non-equilibrium spin systems:

```
integer iL,iR,iU,iO,iS,iN,edelz
double precision tt
...                              ! load spin orientations
edelz=iL+iR+iU+iO+iS+iN          ! loop kernel
BF= 0.5d0*(1.d0+tanh(edelz/tt))
```

The last two lines are executed in a loop that accounts for nearly the whole runtime of the application. The integer variables store spin orientations (up or down, i.e. -1 or $+1$, respectively), so the `edelz` variable only takes integer values in the range $\{-6,\ldots,+6\}$. The `tanh()` function is one of those operations that take vast amounts of time (at least tens of cycles), even if implemented in hardware. In the case described, however, it is easy to eliminate the `tanh()` call completely by tabulating the function over the range of arguments required, assuming that `tt` does not change its value so that the table does only have to be set up once:

```
double precision tanh_table(-6:6)
integer iL,iR,iU,iO,iS,iN, edelz
double precision, tt
...
do i=-6,6                         ! do this once
  tanh_table(i) = tanh(dble(i)/tt)
```

```
enddo
...
edelz=iL+iR+iU+iO+iS+iN          ! loop kernel
BF= 0.5d0*(1.d0+tanh_table(edelz))
```

The table lookup is performed at virtually no cost compared to the `tanh()` evaluation since the table will, due to its small size and frequent use, be available in L1 cache at access latencies of a few CPU cycles.

27.1.1.3 Shrink the Working Set!

The working set of a code is the amount of memory it uses (i.e. actually touches) in the course of a calculation. In general, shrinking the working set by whatever means is a good thing because it raises the probability for cache hits. If and how this can be achieved and whether it pays off performance-wise depends heavily on the algorithm and its implementation, of course. In the above example, the original code used standard four-byte integers to store the spin orientations. The working set was thus much larger than the L2 cache of any processor. By changing the array definitions to use `integer*1` for the spin variables, the working set could be reduced by nearly a factor of four, and became comparable to cache size.

Many recent microprocessor designs have instruction set extensions for integer and floating-point SIMD operations (see also Sect. 26.1.4) that allow the concurrent execution of arithmetic operations on a wide register that can hold, e.g., two DP or four SP floating-point words. Although vector processors also use SIMD instructions and the use of SIMD in microprocessors is often coined vectorization, it is more similar to the multi-track property of modern vector systems. Generally speaking, a vectorizable loop in this context will run faster if more operations can be performed with a single instruction, i.e. the size of the data type should be as small as possible. Switching from DP to SP data could result in up to a twofold speedup, with the additional benefit that more items fit into the cache.

Consider, however, that not all microprocessors can handle small types efficiently. Using byte-size integers for instance could result in very ineffective code that actually works on larger word sizes but extracts the byte-sized data by mask and shift operations.

27.1.1.4 Eliminate Common Subexpressions!

Common subexpression elimination is an optimization that is often considered a task for compilers. Basically one tries to save time by pre-calculating parts of complex expressions and assigning them to temporary variables before a loop starts:

```
! inefficient
do i=1,N
  A(i)=A(i)+s+r*sin(x)
enddo
```

$\longrightarrow$

```
tmp=s+r*sin(x)
do i=1,N
  A(i)=A(i)+tmp
enddo
```

A lot of compute time can be saved by this optimization, especially where strong operations (like `sin()`) are involved. Although it may happen that subexpressions are obstructed by other code and not easily recognizable, compilers are in principle able to detect this situation. They will however often refrain from pulling the subexpression out of the loop except with very aggressive optimizations turned on. The reason for this is the well-known non-associativity of FP operations: If floating-point accuracy is to be maintained compared to non-optimized code, associativity rules must not be used and it is left to the programmer to decide whether it is safe to regroup expressions by hand.

27.1.1.5 Avoid Conditionals in Tight Loops!

Tight loops, i.e. loops that have few operations in them, are typical candidates for software pipelining (see Sect. 26.1.3.1), loop unrolling and other optimization techniques (see below). If for some reason compiler optimization fails or is inefficient, performance will suffer. This can easily happen if the loop body contains conditional branches:

```
do j=1,N
  do i=1,N
    if(i.ge.j) then
      sign=1.d0
    else if(i.lt.j) then
      sign=-1.d0
    else
      sign=0.d0
    endif
    C(j) = C(j) + sign * A(i,j) * B(i)
  enddo
enddo
```

In this multiplication of a matrix with a vector, the upper and lower triangular parts get different signs and the diagonal is ignored. The `if` statement serves to decide about which factor to use. Each time a corresponding conditional branch is encountered by the processor, some branch prediction logic tries to guess the most probable outcome of the test before the result is actually available, based on statistical methods. The instructions along the chosen path are then fetched, decoded, and generally fed into the pipeline. If the anticipation turns out to be false (this is called a mispredicted branch or branch miss), the pipeline has to be flushed back to the position of the branch, implying many lost cycles. Furthermore, the compiler refrains from doing advanced optimizations like loop unrolling (see Sect. 27.1.3.2).

Fortunately the loop nest can be transformed so that all `if` statements vanish:

```
do j=1,N
  do i=j+1,N
    C(j) = C(j) + A(i,j) * B(i)
  enddo
```

```
  enddo
  do j=1,N
    do i=1,j-1
      C(j) = C(j) - A(i,j) * B(i)
    enddo
  enddo
```

By using two different variants of the inner loop, the conditional has virtually been moved outside. One should add that there is more optimization potential in this loop nest. Please consider the section on data access below for more information.

27.1.1.6 Use Compiler Logs!

The previous sections have pointed out that the compiler is a crucial component in writing efficient code. It is very easy to hide important information from the compiler, forcing it to give up optimization at an early stage. In order to make the decisions of the compiler's intelligence available to the user, many compilers offer options to generate annotated source code listings or at least logs that describe in some detail what optimizations were performed. Listing 27.1 shows an example for a compiler annotation regarding a standard vector triad loop as in listing 26.1. Unfortunately, not all compilers have the ability to write such comprehensive code annotations and users are often left with guesswork.

27.1.2 Data Access

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. As many applications in science and engineering consist of loop-based code that

Listing 27.1. Compiler log for a software pipelined triad loop

```
#<swps> 16383 estimated iterations before pipelining
#<swps>     4 unrollings before pipelining
#<swps>    20 cycles per 4 iterations
#<swps>     8 flops        ( 20% of peak) (madds count as 2)
#<swps>     4 flops        ( 10% of peak) (madds count as 1)
#<swps>     4 madds        ( 20% of peak)
#<swps>    16 mem refs     ( 80% of peak)
#<swps>     5 integer ops  ( 12% of peak)
#<swps>    25 instructions ( 31% of peak)
#<swps>     2 short trip threshold
#<swps>    13 integer registers used.
#<swps>    17 float registers used.
```

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. Any optimization attempt should therefore 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.

27.1.2.1 Balance and Lightspeed Estimates

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 is the ratio of possible memory bandwidth in GWords/sec to peak performance in GFlops/sec:

$$B_{\mathrm{m}} = \frac{\text{memory bandwidth [GWords/sec]}}{\text{peak performance [GFlops/sec]}} \,. \tag{27.1}$$

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 prefetching and software pipelining. As an example, consider a processor with a clock frequency of 3.2 GHz that can perform at most two flops per cycle and has a memory bandwidth of 6.4 GBytes/sec. This processor would have a machine balance of 0.125 W/F. At the time of writing, typical values of B_{m} lie in the range between 0.1 W/F for commodity microprocessors and 0.5 W/F for top of the line vector computers. Due to the continuously growing DRAM gap and the advent of multi-core designs, machine balance for standard architectures will presumably decrease further in the future. Table 27.1 shows typical balance values for several possible transfer paths.

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

$$B_{\mathrm{c}} = \frac{\text{data traffic volume [Words]}}{\text{floating point operations [Flops]}} \,. \tag{27.2}$$

Table 27.1. Typical balance values for operations limited by different transfer paths

data path	balance
cache	0.5–1.0
machine (memory)	0.05–0.5
interconnect (high speed)	0.01–0.04
interconnect (GBit ethernet)	0.001–0.003
disk	0.001–0.02

Now it is obvious that the expected maximum fraction of peak performance one can expect from a code with balance B_c on a machine with balance B_m is

$$l = \min\left(1, \frac{B_\mathrm{m}}{B_\mathrm{c}}\right) . \tag{27.3}$$

We call this fraction the lightspeed of a code. If $l \simeq 1$, loop 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 (multiplication and addition) in an optimal way. If this is not the case, e.g., when only additions are used, one must introduce a correction term that reflects the ratio of MULT to ADD operations.
- Code is based on double precision floating-point arithmetic. In cases where this is not true, one can easily derive similar, more appropriate metrics (e.g., words per instruction).
- Data transfer and arithmetic overlap perfectly.
- The system is in throughput mode, i.e. latency effects are negligible.

We must emphasize that more advanced strategies for performance modeling do exist and refer to the literature [1, 2].

As an example consider the standard vector triad benchmark introduced in Sect. 26.1.5. The kernel loop,

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

features 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 $B_\mathrm{c} = (3+1)/2 = 2$. On a CPU with machine balance $B_\mathrm{m} = 0.1$, we can then expect a lightspeed ratio of 0.05, i.e. 5 % of peak.

Standard cache-based microprocessors usually feature an outermost cache level with write-back strategy. As explained in Sect. 26.1.5, cache line read for ownership (RFO) is then required 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 $l_\mathrm{RFO} = 0.04$.

27.1.2.2 Storage Order of Multi-Dimensional Arrays

Multi-dimensional 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 multi-dimensional data structure must be matched to the order in which code loads and stores data so that spatial and temporal locality can

be employed. In Sect. 26.1.5 it was shown that strided access to a one-dimensional array reduces spatial locality, leading to low utilization of the available bandwidth. When dealing with multi-dimensional arrays, those access patterns can be generated quite naturally:

Stride-N access

```
do i=1,N
  do j=1,N
    A(i,j) = i*j
  enddo
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 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 Fortran follows the so-called column major order whereas C follows row major order for multi-dimensional arrays (see Fig. 27.1). Although mathematically insignificant, the distinction must be kept in mind when optimizing for data access.

27.1.2.3 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, $A = B^{\mathrm{T}}$, 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:

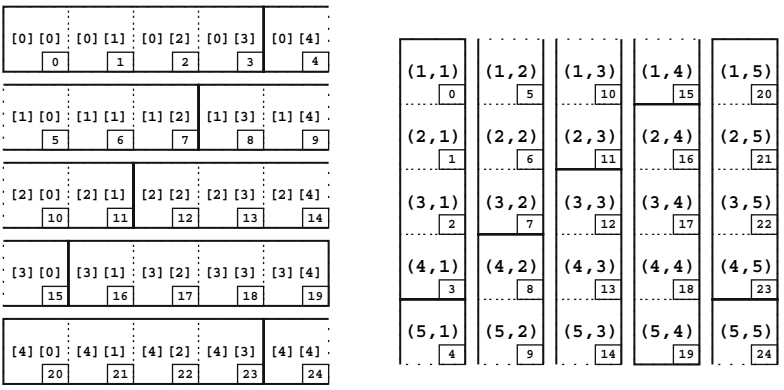

Fig. 27.1. Row major order (**left**) and column major order (**right**) storage schemes for matrices. The small numbers indicate the offset of the element with respect to the starting address of the array. Solid frames symbolize cache lines

```
do i=1,N
  do j=1,N
    A(i,j) = B(j,i)
  enddo
enddo
```

Write access to matrix A is strided (see Fig. 27.2). Due to RFO 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.
- If the matrices are too large to fit into cache but still

$$NL_c \lesssim C\ , \tag{27.4}$$

  the strided access to A is insignificant because all stores performed during a complete traversal of a row that cause a write miss start a cache line RFO. Those lines are most probably still in cache for the next $L_c - 1$ rows, alleviating the effect of strided write (spatial locality). Effective bandwidth should be of the order of the processor's memory bandwidth.
- If N is even larger so that $NL_c \gtrsim C$, each store to A causes a cache miss and a subsequent RFO. 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.

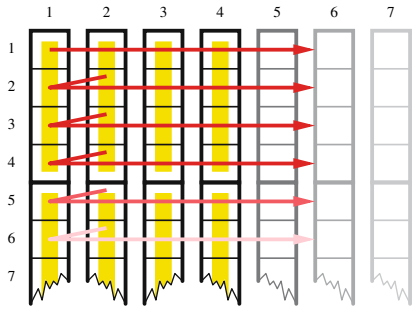

Fig. 27.2. Cache line traversal for vanilla matrix transpose (strided store stream, column major order). If the leading matrix dimension is a multiple of the cache line size, each column starts on a line boundary

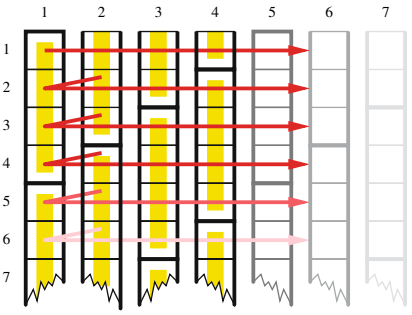

Fig. 27.3. Cache line traversal for padded matrix transpose. Successive iterations hit different cache lines

The vanilla graph in Fig. 27.4 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 cache. This is because the L1 cache on the considered architecture is of write-through type, i.e. the L2 cache is always updated on a write, regardless whether there was an L1 hit or miss. The RFO transactions between the two caches hence waste the major part of 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

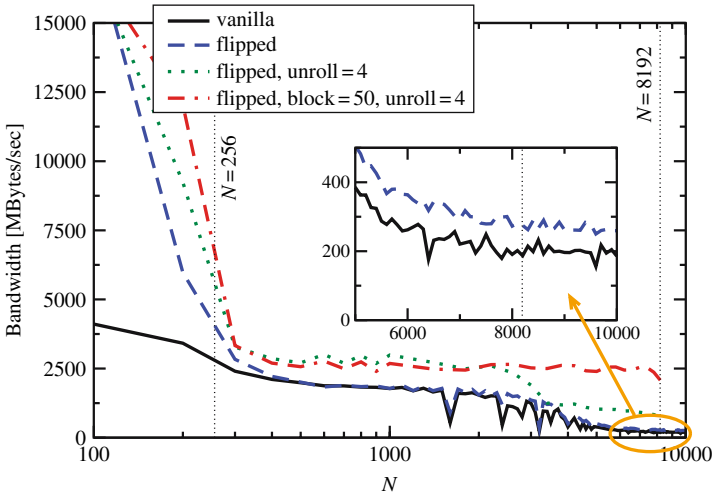

Fig. 27.4. Performance (effective bandwidth) for different implementations of the dense matrix transpose on a modern microprocessor with 1 MByte of L2 cache. The $N = 256$ and $N = 8192$ lines indicate the positions where the matrices fit into cache and where N cache lines fit into cache, respectively. (Intel Xeon/Nocona 3.2 Ghz)

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 50 % is usually attainable, so there must be a factor that further reduces available bandwidth. This factor is the translation look-aside buffer (TLB) that 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 it must be expected that this cache level cannot be used with optimal performance. 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 L2 cache, this penalty reduces effective bandwidth significantly because every TLB miss leads to an additional access latency of at least 57 processor cycles. 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 half a 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 "hit the floor".

At a cache line length of 16 words (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 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 here that performance predictions based on architectural specifications do work in many, but not in all cases, especially on commodity systems where factors like chip sets, 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 the inset in Fig. 27.4, flipped graph), albeit very noticeable, falls short of this expectation. One possible reason for this could be a slightly better effectivity of the memory interface with strided writes.

In general, the performance graphs in Fig. 27.4 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 Fig. 27.5) 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 Sect. 26.1.5.2, strided memory access leads to thrashing

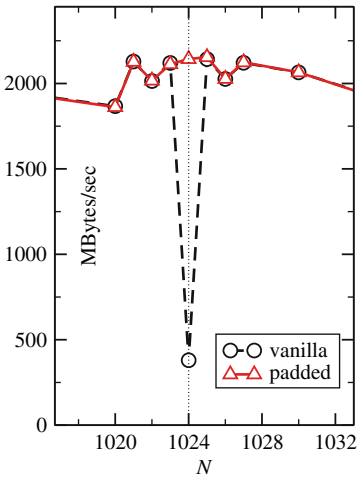

Fig. 27.5. Cache thrashing for unfavorable choice of array dimensions (dashed). Padding removes thrashing completely (solid)

when successive iterations hit the same (set of) cache line(s) because of insufficient associativity. Fig. 27.2 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 L_c words, the effective cache size is

$$C_{\text{eff}} = L_c \max\left(1, \frac{C}{N}\right) . \tag{27.5}$$

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 1024×1024, enlarging the leading dimension by p (called padding) to get `A(1024+`p`,1024)` leads to 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 Fig. 27.3). In Fig. 27.5 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 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 will be considered in the following sections.

27.1.3 Data Access Optimizations and Classification of Algorithms

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.

27.1.3.1 $O(N)/O(N)$

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:

```
do i=1,N                                  ! optimized
  A(i) = B(i) + C(i)                      do i=1,N
enddo                    loop fusion        A(i) = B(i) + C(i)
do i=1,N                 ---------->      ! save a load for B(i)
  Z(i) = B(i) + E(i)                      Z(i) = B(i) + E(i)
enddo                                     enddo
```

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 RFOs). 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 RFO 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.

27.1.3.2 $O(N^2)/O(N^2)$

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 multiplication, 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 a constant factor of improvement. Consider dense matrix-vector multiplication (MVM):

```
do i=1,N
  tmp = C(i)
  do j=1,N
    tmp = tmp + A(j,i) * B(j)
  enddo
  C(i) = tmp
enddo
```

This code has a balance of 1 (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. 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. Obviously, one has 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:

```
! remainder loop
do r=1,mod(N,m)
  do j=1,N
    C(r) = C(r) + A(j,r) * B(j)
  enddo
enddo
! main loop
do i=r,N,m
  do j=1,N
    C(i) = C(i) + A(j,i) * B(j)
  enddo
  do j=1,N
    C(i+1) = C(i+1) + A(j,i+1) * B(j)
  enddo
  ! m times
  ...
  do j=1,N
    C(i+m-1) = C(i+m-1) + A(j,i+m-1) * B(j)
  enddo
enddo
```

The remainder loop is obviously 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:

```
! remainder loop ignored
do i=1,N,m
  do j=1,N
    C(i) = C(i) + A(j,i) * B(j)
    C(i+1) = C(i+1) + A(j,i+1) * B(j)
    !  m times
    ...
    C(i+m-1) = C(i+m-1) + A(j,i+m-1) * B(j)
  enddo
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 $N^2(1+1/m)+N$. Fig. 27.6 shows the memory access pattern for vanilla and 2-way unrolled dense MVM.

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. Again, compiler logs 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 non-portable.

The matrix transpose code from the previous section is another example for a problem of $O(N^2)/O(N^2)$ type, although in contrast to dense MVM there is no direct opportunity for saving on memory traffic; both matrices have to be read

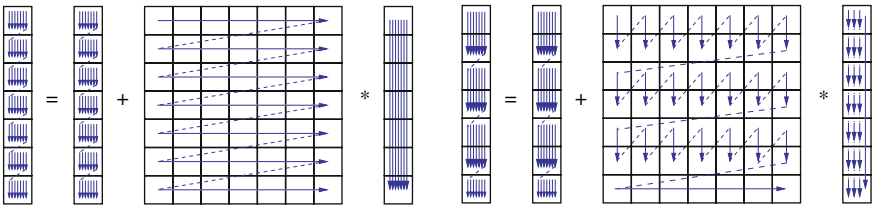

Fig. 27.6. Vanilla (**left**) and 2-way unrolled (**right**) dense matrix vector multiplication. The remainder loop is only a single (outer) iteration in this example

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 Fig. 27.4):

```
do j=1,N,m
  do i=1,N
    A(i,j)     = B(j,i)
    A(i,j+1)   = B(j+1,i)
    ...
    A(i,j+m-1) = B(j+m-1,i)
  enddo
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. The left picture in Fig. 27.7 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 loop nest of depth d, blocking introduces up to d additional outer loop levels that cut the original inner loops into chunks:

```
do jj=1,N,b
  jstart=jj; jend=jj+b-1
  do ii=1,N,b
    istart=ii; iend=ii+b-1
    do j=jstart,jend,m
      do i=istart,iend
        a(i,j) = b(j,i)
        a(i,j+1) = b(j+1,i)
        ...
        a(i,j+m-1) = b(j+m-1,i)
      enddo
    enddo
  enddo
enddo
```

In this example we have used 2D blocking with identical blocking factors b for both loops in addition to m-way unroll and jam. Obviously, this change does not

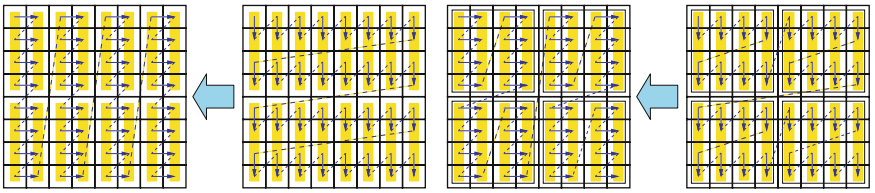

Fig. 27.7. Two-way unrolled (**left**) and blocked/unrolled (**right**) flipped matrix transpose, i.e. with strided load

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 the right picture in Fig. 27.7 which shows a combination of two-way unrolling and 4×4 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 Fig. 27.4 demonstrates that 50×50 blocking combined with 4-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 cache level to block for depends on the operations performed and there is no general recommendation.

27.1.3.3 $O(N^3)/O(N^2)$

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 usually 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 contribution 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:

```
do i=1,N
  do j=1,N
    sum = sum + foo(A(i),B(j))
  enddo
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 block size of b, however,

```
do jj=1,N,b
  jstart=jj; jend=jj+b-1
  do i=1,N
    do j=jstart,jend
      sum = sum + foo(A(i),B(j))
    enddo
  enddo
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 MMM, 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 cache, many processor architectures do not have the capability for sustaining enough loads and stores per cycle to feed the arithmetic units continuously. The once widely used but now outdated MIPS R1X000 family of processors for instance could only sustain one load *or* store operation per cycle, which makes unroll and jam mandatory if the kernel of a loop nest uses more than one stream, especially in cache-bound situations like the blocked $O(N^2)/O(N)$ example above.

Although demonstrated here for educational purpose, 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 MVM (see next section).

27.1.4 Case Study: Sparse Matrix-Vector Multiplication

An interesting real-world application of the blocking and unrolling strategies discussed in the previous sections is the multiplication of a sparse matrix with a vector. It is a key ingredient in most iterative matrix diagonalization algorithms (Lanczos, Davidson, Jacobi-Davidson; see Chap. 18) and usually a performance-limiting factor. A matrix is called sparse if the number of non-zero entries N_{nz} grows linearly with the number of matrix rows N_{r}. Of course, only the non-zeroes are stored at all for efficiency reasons. Sparse MVM (sMVM) is hence an $O(N_{\mathrm{r}})/O(N_{\mathrm{r}})$ problem and inherently memory-bound if N_{r} is reasonably large. Nevertheless, the presence of loop nests enables some significant optimization potential. Fig. 27.8 shows that sMVM generally requires some strided or even indirect addressing of the r.h.s. vector, although there exist matrices for which memory access patterns are much more favorable. In the following we will keep at the general case.

27.1.4.1 Sparse Matrix Storage Schemes

Several different storage schemes for sparse matrices have been developed, some of which are suitable only for special kinds of matrices [3]. Of course, memory access patterns and thus performance characteristics of sMVM depend heavily on the storage scheme used. The two most important and also general formats are CRS (Compressed Row Storage) and JDS (Jagged Diagonals Storage). We will see that CRS is well-suited for cache-based microprocessors while JDS supports dependency and loop structures that are favorable on vector systems.

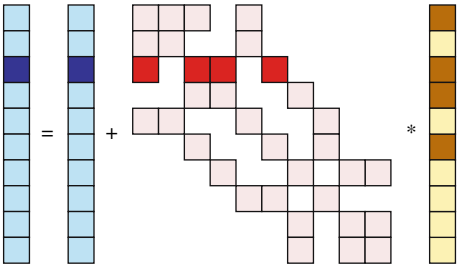

Fig. 27.8. Sparse matrix-vector multiplication. Dark elements visualize entries involved in updating a single l.h.s. element. Unless the sparse matrix rows have no gaps between the first and last non-zero elements, some indirect addressing of the r.h.s. vector is inevitable

In CRS, an array `val` of length N_{nz} is used to store all non-zeroes 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 non-zero element in `val`, and `row_ptr` contains the indices at which new rows start in `val` (see Fig. 27.9). The basic code to perform a MVM using this format is quite simple:

```
do i = 1,Nr
  do j = row_ptr(i), row_ptr(i+1) - 1
    c(i) = c(i) + val(j) * b(col_idx(j))
  enddo
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 memory.
- The non-zeroes in `val` are accessed with stride one.
- As expected, the r.h.s. vector `b` is accessed indirectly. This may however not be a serious performance problem depending on the exact structure of the matrix. If the non-zeroes are concentrated mainly around the diagonal, there will even be considerable spatial and/or temporal locality.
- $B_c = 5/4$ if the integer load to `col_idx` is counted with four bytes.

Some of those points will be of importance later when we demonstrate parallel sMVM (see Sect. 27.2.2).

JDS requires some rearrangement of the matrix entries beyond simple zero elimination. First, all zeroes are eliminated from the matrix rows and the non-zeroes are shifted to the left. Then the matrix rows are sorted by descending number of non-zeroes 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

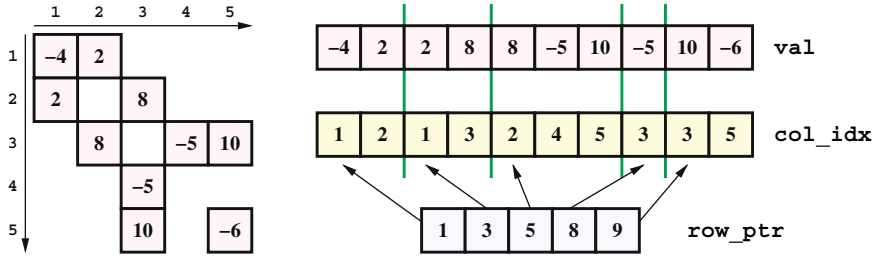

Fig. 27.9. CRS sparse matrix storage format

sparse matrix from left top to right bottom (see Fig. 27.10). For each non-zero the original column index is stored in `col_idx` just like in the CRS. In order to have the same element order on the r.h.s. and l.h.s. 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:

```
do diag=1, Nj
  diagLen = jd_ptr(diag+1) - jd_ptr(diag)
  offset = jd_ptr(diag)
  do i=1, diagLen
    c(i) = c(i) + val(offset+i) * b(col_idx(offset+i))
  enddo
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).

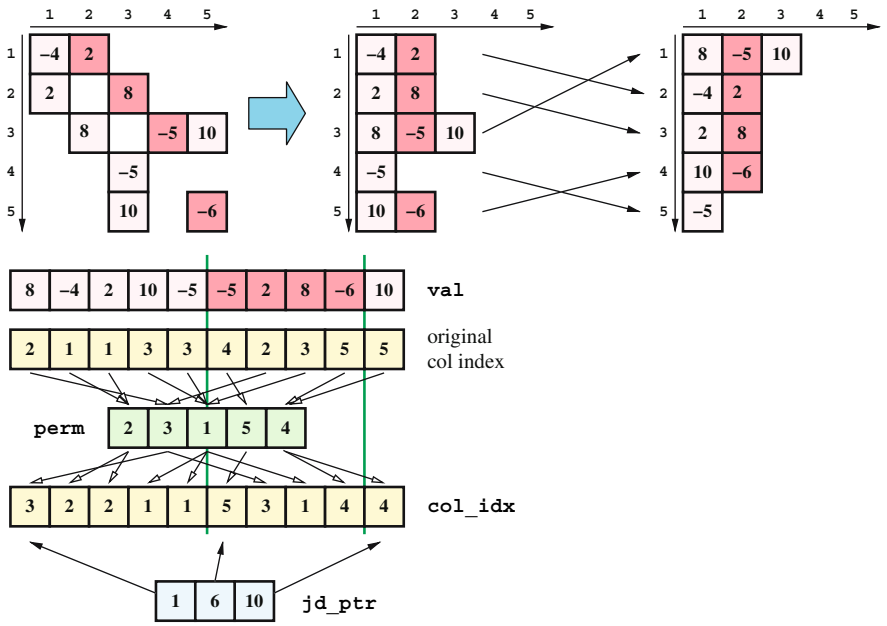

Fig. 27.10. JDS sparse matrix storage format. The permutation map is also applied to the column index array. One of the jagged diagonals is marked

- The result vector is loaded multiple times (at least partially) from memory, so there might be some optimization potential.
- The non-zeroes in `val` are accessed with stride one.
- The r.h.s. 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 r.h.s. vector are permuted.
- $B_c = 9/4$ 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.

27.1.4.2 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 Fig. 27.11; the remainder loop is omitted as usual):

```
do diag=1,Nj,2  ! 2-way unroll & jam
  diagLen = min( (jd_ptr(diag+1)-jd_ptr(diag)) ,\
                 (jd_ptr(diag+2)-jd_ptr(diag+1)) )
  offset1 = jd_ptr(diag)
  offset2 = jd_ptr(diag+1)
  do i=1, diagLen
    c(i) = c(i)+val(offset1+i)*b(col_idx(offset1+i))
    c(i) = c(i)+val(offset2+i)*b(col_idx(offset2+i))
  enddo
  ! peeled-off iterations
  offset1 = jd_ptr(diag)
  do i=(diagLen+1),(jd_ptr(diag+1)-jd_ptr(diag))
    c(i) = c(i)+val(offset1+i)*b(col_idx(offset1+i))
  enddo
enddo
```

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

$$B_c = \frac{1}{m} + \frac{5}{4}\,.$$

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 Fig. 27.12):

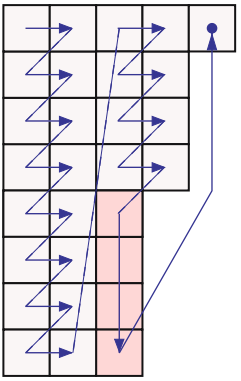

Fig. 27.11. JDS matrix traversal with 2-way unroll and jam and loop peeling. The peeled iterations are marked

```
! loop over blocks
do ib=1, Nr, bl
  block_start = ib
  block_end   = min(ib+bl-1, Nr)
  ! loop over diagonals in one block
  do diag=1, Nj
    diagLen = jd_ptr(diag+1)-jd_ptr(diag)
    offset = jd_ptr(diag)
    if(diagLen .ge. block_start) then
      ! standard JDS sMVM kernel
      do i=block_start, min(block_end,diagLen)
        c(i) = c(i)+val(offset+i)*b(col_idx(offset+i))
      enddo
    endif
  enddo
enddo
```

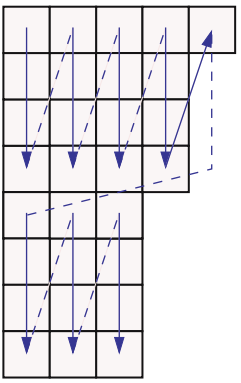

Fig. 27.12. JDS matrix traversal with 4-way loop blocking

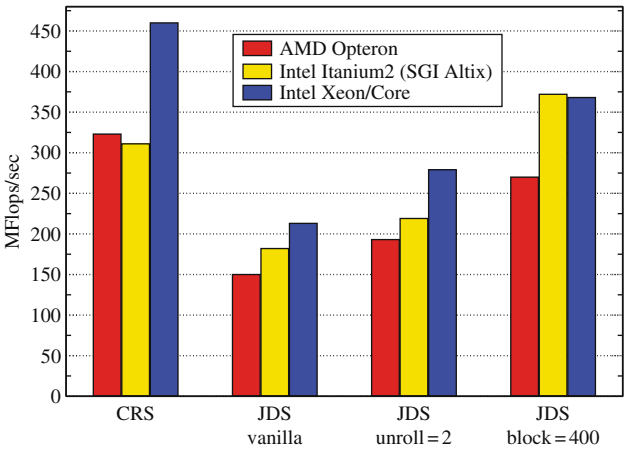

Fig. 27.13. Performance comparison of sparse MVM codes with different optimizations. A matrix with 1.7×10^7 unknowns and 20 jagged diagonals was chosen. The blocking size of 400 has proven to be optimal for a wide range of architectures

With this optimization the result vector is effectively loaded only once from memory if the block size `bl` 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.

Fig. 27.13 shows a performance comparison of CRS and plain, 2-way unrolled and blocked ($b = 400$) JDS sMVM on three different architectures. 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, however, shows mediocre performance for CRS and tops at the blocked JDS version. This architecture can not 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.

27.2 Shared-Memory Parallelization

OpenMP seems to be the easiest way to write parallel programs as it features a simple, directive-based interface and incremental parallelization, meaning that the loops of a program can be tackled one by one without major code restructuring. It turns out, however, that getting a truly scalable OpenMP program is a significant undertaking in all but the most trivial cases. This section pinpoints some of the performance problems that can arise with shared-memory programming and how they can be circumvented. We then turn to the OpenMP parallelization of the sparse MVM code that has been demonstrated in the previous sections.

27.2.1 Performance Pitfalls

Like any other parallelization method, OpenMP is prone to the standard problems of parallel programming: Serial fraction (Amdahl's law) and load imbalance, both introduced in Sect. 26.2.

An overabundance of serial code can easily arise when critical sections become out of hand. If all but one threads continuously wait for a critical section to become available, the program is effectively serialized. This can be circumvented by employing finer control on shared resources using named critical sections or OpenMP locks. Sometimes it may even be useful to supply thread-local copies of otherwise shared data that may be pulled together by a reduction operation at the end of a parallel region. The load imbalance problem can often be solved by choosing a different OpenMP scheduling strategy (see Sect. 26.2.4.4).

There are, however, very specific performance problems that are inherently connected to shared-memory programming in general and OpenMP in particular.

27.2.1.1 OpenMP Overhead

Whenever a parallel region is started or stopped or a parallel loop is initiated or ended, there is some non-negligible overhead involved. Threads must be spawned or at least woken up from an idle state, the size of the work packages (chunks) for each thread must be determined, and in the case of dynamic or guided scheduling schemes each thread that becomes available must be supplied with a new chunk to work on. Generally, the overhead caused by the start of a parallel region consists of a (large) constant part and a part that is proportional to the number of threads. There are vast differences from system to system as to how large this overhead can be, but it is generally of the order of at least hundreds if not thousands of CPU cycles. If the programmer follows some simple guidelines, the adverse effects of OpenMP overhead can be much reduced:

– Avoid parallelizing short, tight loops. If the loop body does not contain much work, i.e. if each iteration executes in a very short time, OpenMP loop overhead will lead to very bad performance. It is often beneficial to execute a serial version if the loop count is below some threshold. The OpenMP `IF` clause helps with this:

```
!$OMP PARALLEL DO IF(N>10000)
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
!$OMP END PARALLEL DO
```

Fig. 27.14 shows a comparison of vector triad data in the purely serial case and with one and four OpenMP threads, respectively. The presence of OpenMP causes overhead at small N even if only a single thread is used. Using the `IF` clause leads to an optimal combination of threaded and serial loop versions if

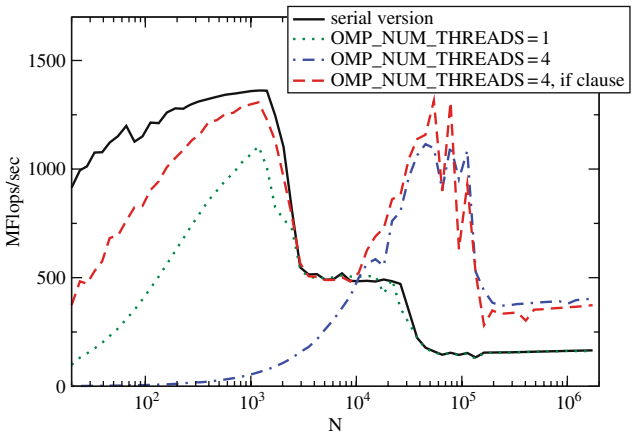

Fig. 27.14. OpenMP overhead and the benefits of the `IF(N>10000)` clause for the vector triad benchmark. Note the impact of aggregate cache size on the position of the performance breakdown from L2 to memory. (AMD Opteron 2.0 GHz)

the threshold is chosen appropriately, and is hence mandatory when large loop lengths cannot be guaranteed.

As a side-note, there is another harmful effect of short loop lengths: If the number of iterations is comparable to the number of threads, load imbalance may cause bad scalability.

– In loop nests, parallelize on a level as far out as possible. This is inherently connected to the previous advice. Parallelizing inner loop levels leads to increased OpenMP overhead because a team of threads is spawned or woken up multiple times.
– Be aware that most OpenMP work-sharing constructs (including `OMP DO` and `END DO`) insert automatic barriers at the end so that all threads have completed their share of work before anything after the construct is executed. In cases where this is not required, a `NOWAIT` clause removes the implicit barrier:

```
!$OMP PARALLEL
!$OMP DO
  do i=1,N
    A(i) = func1(B(i))
  enddo
!$OMP END DO NOWAIT
! still in parallel region here. do more work:
!$OMP CRITICAL
  CNT = CNT + 1
!$OMP END CRITICAL
!$OMP END PARALLEL
```

There is also an implicit barrier at the end of a parallel region that cannot be removed. In general, implicit barriers add to synchronization overhead like critical regions, but they are often required to protect from race conditions.

27.2.1.2 False Sharing

The hardware-based cache coherence mechanisms described in Sect. 26.2.4 make the use of caches in a shared-memory system transparent to the programmer. In some cases, however, cache coherence traffic can throttle performance to very low levels. This happens if the same cache line is modified continuously by a group of threads so that the cache coherence logic is forced to evict and reload it in rapid succession. As an example, consider a program fragment that calculates a histogram over the values in some large integer array A that are all in the range $\{1, \ldots, 8\}$:

```
integer, dimension(8) :: S
integer IND
S = 0
do i=1,N
  IND = A(i)
  S(IND) = S(IND) + 1
enddo
```

In a straightforward parallelization attempt one would probably go about and make S two-dimensional, reserving space for the local histogram of each thread:

```
  integer, dimension(:,:), allocatable :: S
  integer IND,ID,NT
!$OMP PARALLEL PRIVATE(ID,IND)
!$OMP SINGLE
  NT = omp_get_num_threads()
  allocate(S(0:NT,8))
  S = 0
!$OMP END SINGLE
 ID = omp_get_thread_num() + 1
!$OMP DO
  do i=1,N
   IND = A(i)
   S(ID,IND) = S(ID,IND) + 1
  enddo
!$OMP END DO NOWAIT
  ! calculate complete histogram
!$OMP CRITICAL
  do j=1,8
  S(0,j) = S(0,j) + S(ID,j)
  enddo
!$OMP END CRITICAL
!$OMP END PARALLEL
```

The loop starting at line 18 collects the partial results of all threads. Although this is a valid OpenMP program, it will not run faster but much more slowly when using four threads instead of one. The reason is that the two-dimensional array `S` contains all the histogram data from all threads. With four threads these are 160 bytes, less than two cache lines on most processors. On each histogram update to `S` in line 10, the writing CPU must gain exclusive ownership of one of the two cache lines, i.e. every write leads to a cache miss and subsequent coherence traffic. Compared to the situation in the serial case where `S` fits into the cache of a single CPU, this will result in disastrous performance.

One should add that false sharing can be eliminated in simple cases by the standard register optimizations of the compiler. If the crucial update operation can be performed to a register whose contents are only written out at the end of the loop, no write misses turn up. This is not possible in the above example, however, because of the computed second index to `S` in line 10.

Getting rid of false sharing by manual optimization is often a simple task once the problem has been identified. A standard technique is array padding, i.e. insertion of a suitable amount of space between memory locations that get updated by different threads. In the histogram example above, an even more painless solution exists in the form of data privatization: On entry to the parallel region, each thread gets its own *local* copy of the histogram array in its own stack space. It is very unlikely that those different instances will occupy the same cache line, so false sharing is not a problem. Moreover, the code is simplified and made equivalent with the serial version by using the `REDUCTION` clause introduced in Sect. 26.2.4.4:

```
    integer, dimension(8) :: S
    integer IND
    S=0
!$OMP PARALLEL DO PRIVATE(IND) REDUCTION(+:S)
    do i=1,N
      IND = A(i)
      S(IND) = S(IND) + 1
    enddo
!$OMP EMD PARALLEL DO
```

Setting `S` to zero is only required for serial equivalence as the reduction clause automatically initializes the variables in question with appropriate starting values. We must add that OpenMP reduction to arrays in Fortran does not work for allocatable, pointer or assumed size types.

27.2.2 Case Study: Parallel Sparse Matrix-Vector Multiplication

As an interesting application of OpenMP to a nontrivial problem we now extend the considerations on sparse MVM data layout and optimization by parallelizing the CRS and JDS matrix-vector multiplication codes from Sect. 27.1.4.

No matter which of the two storage formats is chosen, the general parallelization approach is always the same: In both cases there is a parallelizable loop that

calculates successive elements (or blocks of elements) of the result vector (see Fig. 27.15). For the CRS matrix format, this principle can be applied in a straightforward manner:

```
!$OMP PARALLEL DO PRIVATE(j)[1]
  do i = 1,N_r
    do j = row_ptr(i), row_ptr(i+1) - 1
      c(i) = c(i) + val(j) * b(col_idx(j))
    enddo
  enddo
!$OMP END PARALLEL DO
```

Due to the long outer loop, OpenMP overhead is usually not a problem here. Depending on the concrete form of the matrix, however, some loop imbalance might occur if very short or very long matrix rows are clustered at some regions. A different kind of OpenMP scheduling strategy like `DYNAMIC` or `GUIDED` might help in this situation.

The vanilla JDS sMVM is also parallelized easily:

```
!$OMP PARALLEL PRIVATE(diag,diagLen,offset)
  do diag=1, N_j
    diagLen = jd_ptr(diag+1) - jd_ptr(diag)
    offset = jd_ptr(diag)
!$OMP DO
    do i=1, diagLen
      c(i) = c(i) + val(offset+i) * b(col_idx(offset+i))
    enddo
!$OMP END DO
  enddo
```

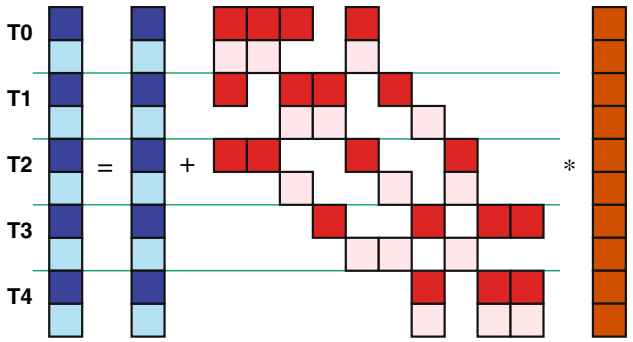

Fig. 27.15. Parallelization approach for sparse MVM (five threads). All marked elements are handled in a single iteration of the parallelized loop. The r.h.s. vector is accessed by all threads

[1] The privatization of inner loop indices in the lexical extent of a parallel outer loop is not required in Fortran, but it is in C/C++ [4].

```
!$OMP END PARALLEL
```

The parallel loop is the inner loop in this case, but there is no OpenMP overhead problem as the loop count is large. Moreover, in contrast to the parallel CRS version, there is no load imbalance because all inner loop iterations contain the same amount of work. All this would look like an ideal situation were it not for the bad code balance of vanilla JDS sMVM. However, the unrolled and blocked versions can be equally well parallelized. For the blocked code (see Fig. 27.12), the outer loop over all blocks is a natural candidate:

```
!$OMP DO PARALLEL DO PRIVATE(block_start,block_end,i,diag,
!$OMP& diagLen,offset)
  do ib=1,Nr,b
    block_start = ib
    block_end   = min(ib+b-1,Nr)
    do diag=1,Nj
      diagLen = jd_ptr(diag+1)-jd_ptr(diag)
      offset  = jd_ptr(diag)
      if(diagLen .ge. block_start) then
        do i=block_start, min(block_end,diagLen)
          c(i) = c(i)+val(offset+i)*b(col_idx(offset+i))
        enddo
      endif
    enddo
  enddo
!$OMP END PARALLEL DO
```

This version has even got less OpenMP overhead because the `DO` directive is on the outermost loop. Unfortunately, there is more potential for load imbalance because of the matrix rows being sorted for size. But as the dependence of workload on loop index is roughly predictable, a static schedule with a chunk size of one can remedy most of this effect.

Fig. 27.16 shows performance and scaling behavior of the parallel CRS and blocked JDS versions on three different architectures. In all cases, the code was run on as few locality domains or sockets as possible, i.e. first filling one locality domain or socket before going to the next. On the ccNUMA systems (Altix and Opterons, equivalent to the block diagrams in Figs. 26.23 and 26.24), the performance characteristics with growing CPU number is obviously fundamentally different from the UMA system (Xeon/Core node like in Fig. 26.22). Both code versions seem to be extremely unsuitable for ccNUMA. Only the UMA node shows the expected behavior of strong bandwidth saturation at 2 threads and significant speedup when the second socket gets used (additional bandwidth due to second FSB).

The reason for the failure of ccNUMA to deliver the expected bandwidth lies in our ignorance of a necessary prerequisite for scalability that we have not honored yet: Correct data and thread placement for access locality.

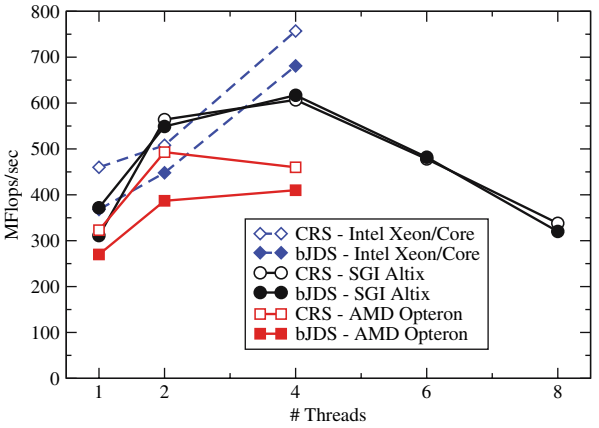

Fig. 27.16. Performance and strong scaling for straightforward OpenMP parallelization of sparse MVM on three different architectures, comparing CRS (open symbols) and blocked JDS (closed symbols) variants. The Intel Xeon/Core system (dashed) is of UMA type, the other two systems are ccNUMA

27.2.3 Locality of Access on ccNUMA

It was mentioned already in the section on ccNUMA architecture that locality and congestion problems (see Figs. 27.17 and 27.18) tend to turn up when threads/processes and their data are not carefully placed across the locality domains of a ccNUMA system. Unfortunately, the current OpenMP standard does not refer to placement at all and it is up to the programmer to use the tools that system builders provide.

The placement problem has two dimensions: First, one has to make sure that memory gets mapped into the locality domains of processors that actually access them. This minimizes NUMA traffic across the network. Second, threads or processes must be "pinned" to those CPUs which had originally mapped their memory regions in order not to lose locality of access. In this context, *mapping* means that a page table entry is set up which describes the association of a physical with a vir-

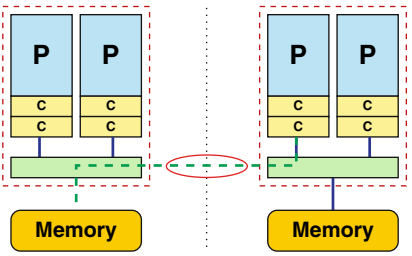

Fig. 27.17. Locality problem on a ccNUMA system. Memory pages got mapped into a locality domain that is not connected to the accessing processor, leading to NUMA traffic

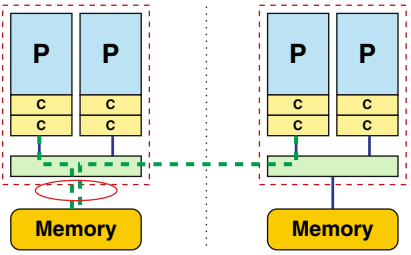

Fig. 27.18. Congestion problem on a ccNUMA system. Even if the network is very fast, a single locality domain can usually not saturate the bandwidth demands from concurrent local and non-local accesses

tual memory page. Consequently, locality of access in ccNUMA systems is always followed on the page level, with typical page sizes of (commonly) 4 kB or (more rarely) 16 kB, sometimes larger. Hence strict locality may be hard to implement with working sets that only encompass a few pages.

27.2.3.1 Ensuring Locality of Memory Access

Fortunately, the initial mapping requirement can be enforced in a portable manner on all current ccNUMA architectures. They support a first touch policy for memory pages: A page gets mapped into the locality domain of the processor that first reads or writes to it. Merely allocating memory is not sufficient (and using `calloc()` in C will most probably be counterproductive). It is therefore the data initialization code that deserves attention on ccNUMA:

```
integer,parameter::N=1000000
double precision A(N), B(N)


! executed on single
! locality domain
READ(1000) A
! congestion problem
!$OMP PARALLEL DO
  do i = 1, N
    B(i) = func(A(i))
  enddo
!$OMP END PARALLEL DO
```

→

```
integer,parameter::N=1000000
double precision A(N), B(N)
!$OMP PARALLEL DO
  do i=1,N
    A(i) = 0.d0
!$OMP END PARALLEL DO
! A is mapped now
READ(1000) A
!$OMP PARALLEL DO
  do i = 1, N
    B(i) = func(A(i))
  enddo
!$OMP END PARALLEL DO
```

On the left, initialization of `A` is done in a serial region using a `READ` staement, so the array data gets mapped to a single locality domain (maybe more if the array is very large). The access to `A` in the parallel loop will then lead to congestion. The version on the right corrects this problem by initializing `A` in parallel, first-touching its elements in the same way they are accessed later. Although the `READ` operation

is still sequential, the data will be distributed across the locality domains. Array `B` does not have to be initialized but will automatically be mapped correctly.

A required condition for this strategy to work is that the OpenMP loop schedules of initialization and work loops are identical and reproducible, i.e. the only possible choice is `STATIC` with a constant chunk size. As the OpenMP standard does not define a default schedule, it is generally a good idea to specify it explicitly on all parallel loops. All current compilers choose `STATIC` by default, though. Of course, the use of a static schedule poses some limits on possible optimizations for eliminating load imbalance. One option is the choice of an appropriate chunk size (as small as possible, but at least several pages).

Unfortunately it is not always at the programmer's discretion how and when data is touched first. In C/C++, global data (including objects) is initialized before the `main()` function even starts. If globals cannot be avoided, properly mapped local copies of global data may be a possible solution, code characteristics in terms of communication vs. calculation permitting [5]. A discussion of some of the problems that emerge from the combination of OpenMP with C++ can be found in [6].

27.2.3.2 ccNUMA Optimization of Sparse MVM

It should now be obvious that the bad scalability of OpenMP-parallelized sparse MVM codes on ccNUMA systems (see Fig. 27.16) is due to congestion that arises because of wrong data placement. By writing parallel initialization loops that exploit first touch mapping policy, scaling can be improved considerably. We will restrict ourselves to CRS here as the strategy is basically the same for JDS. Arrays `c`, `val`, `col_idx`, `row_ptr` and `b` must be initialized in parallel:

```
!$OMP PARALLEL DO
  do i=1,Nr
    row_ptr(i) = 0 ; c(i) = 0.d0 ; b(i) = 0.d0
  enddo
!$OMP END PARALLEL DO
.... ! preset row_ptr array
!$OMP PARALLEL DO PRIVATE(start,end,j)
  do i=1,Nr
    start = row_ptr(i) ; end = row_ptr(i+1)
    do j=start,end-1
      val(j) = 0.d0 ; col_idx(j) = 0
    enddo
  enddo
!$OMP END PARALLEL DO
```

The initialization of `b` is based on the assumption that the non-zeroes of the matrix are roughly clustered around the main diagonal. Depending on the matrix structure it may be hard in practice to perform proper placement for the r.h.s. vector at all.

Fig. 27.19 shows performance data for the same architectures and sMVM codes as in Fig. 27.16 but with appropriate ccNUMA placement. There is no change in

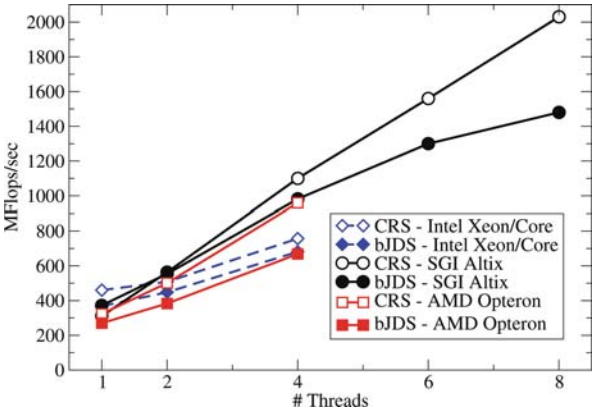

Fig. 27.19. Performance and strong scaling for ccNUMA-optimized OpenMP parallelization of sparse MVM on three different architectures, comparing CRS (open symbols) and blocked JDS (closed symbols) variants. Cf. Fig. 27.16 for performance without proper placement

scalability for the UMA platform, which was to be expected, but also on the ccNUMA systems for up to two threads. The reason is of course that both architectures feature two-processor locality domains which are of UMA type. On four threads and above, the locality optimizations yield dramatically improved performance. Especially for the CRS version scalability is nearly perfect when going from $2n$ to $2(n+1)$ threads (due to bandwidth limitations inside the locality domains, scalability on ccNUMA systems should always be reported with reference to performance on all cores of a locality domain). The JDS variant of the code benefits from the optimizations as well, but falls behind CRS for larger thread numbers. This is because of the permutation map for JDS which makes it hard to place larger portions of the r.h.s. vector into the correct locality domains, leading to increased NUMA traffic.

It should be obvious by now that data placement is of premier importance on ccNUMA architectures, including commonly used two-socket cluster nodes. In principle, ccNUMA features superior scalability for memory-bound codes, but UMA systems are much easier to handle and require no code optimization for locality of access. It is to be expected, though, that ccNUMA designs will prevail in the midterm future.

27.2.3.3 Pinning

One may speculate that the considerations about locality of access on ccNUMA systems from the previous section do not apply for MPI-parallelized code. Indeed, MPI processes have no concept of shared memory. They allocate and first-touch memory pages in their own locality domain *by default*. Operating systems are nowadays capable of maintaining strong affinity between threads and processors, meaning that a thread (or process) will be reluctant to leave the processor it was initially started on. However, it might happen that system processes or interactive load push threads off

their original CPUs. It is not guaranteed that the previous state will be re-established after the disturbance. One indicator for insufficient thread affinity are erratic performance numbers (i.e., varying from run to run). Even on UMA systems insufficient affinity can lead to problems if the UMA node is divided into sections (e.g., sockets with dual-core processors like in Fig. 26.22) that have separate paths to memory and internal shared caches. It may be of advantage to keep neighboring thread IDs on the cores of a socket to exploit the advantage of shared caches. If only one core per socket is used, migration of both threads to the same socket should be avoided if the application is bandwidth-bound.

The programmer can avoid those effects by pinning threads to CPUs. Every operating system has ways of limiting the mobility of threads and processes. Unfortunately, these are by no means portable, but there is always a low-level interface with library calls that access the basic functionality. Under the Linux OS, PLPA [7] can be used for that purpose. The following is a C example that pins each thread to a CPU whose ID corresponds to the thread ID:

```
#include <plpa.h>
...
#pragma omp parallel
{
  plpa_cpu_set_t mask;
  PLPA_CPU_ZERO(&mask);
  int id = omp_get_thread_num();
  PLPA_CPU_SET(id,&mask);
  PLPA_NAME(sched_setaffinity)((pid_t)0, (size_t)32, &mask);
}
```

The `mask` variable is used as a bit mask to identify those CPUs the thread should be restricted to by setting the corresponding bits to one (this could be more than one bit, a feature often called *CPU set*). After this code has executed, no thread will be able to leave its CPU any more.

System vendors often provide high-level interfaces to the pinning or CPU set mechanism. Please consult the system documentation for details.

27.3 Conclusion and Outlook

In this chapter we have presented basic optimization techniques on the processor and the shared-memory level. Although we have mainly used examples from linear algebra for clarity of presentation, the concepts can be applied to all numerical program codes. Although compilers are often surprisingly smart in detecting optimization opportunities, they are also easily deceived by the slightest obstruction of their view on program source. Regrettably, compiler vendors are very reluctant to build tools into their products that facilitate the programmer's work by presenting a clear view on optimizations performed or dropped.

There is one important topic in code optimization that we have neglected for brevity: The start of any serious optimization attempt on a nontrivial application should be the production of a *profile* that identifies the hot spots, i.e. the parts of the code that take the most time to execute. Many tools, free and commercial, exist in this field and more are under development. In which form a programmer should be presented performance data for a parallel run with thousands of processors and how the vast amounts of data can be filtered to extract the important insights is the subject of intense research. Multi-core technologies are adding another dimension to this problem.

References

1. A. Hoisie, O. Lubeck, H. Wassermann, Int. J. High Perform. Comp. Appl. **14**, 330 (2000) 738
2. P.F. Spinnato, G. van Albada, P.M. Sloot, IEEE Trans. Parallel Distrib. Systems **15**(1), 81 (2004) 738
3. R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods* (SIAM, 1994) 750
4. URL `http://www.openmp.org` 760
5. B. Chapman, F. Bregier, A. Patil, A. Prabhakar, Concurrency Comput.: Pract. Exper. **14**, 713 (2002) 764
6. C. Terboven, D. an Mey, in *Proceedings of IWOMP2006 — International Workshop on OpenMP, Reims, France, June 12–15, 2006.* (2006). URL `http://iwomp.univ-reims.fr/cd/papers/TM06.pdf` 764
7. URL `http://www.open-mpi.org/software/plpa/` 766