Chapter 7

Efficient OpenMP programming

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 chapter pinpoints some of the performance problems that can arise with OpenMP shared-memory programming and how they can be circumvented. We then turn to the OpenMP parallelization of the sparse MVM code that has been introduced in Chapter 3.

There is a broad literature on viable optimizations for OpenMP programs [P12, O64]. This chapter can only cover the most relevant basics, but should suffice as a starting point.

7.1 Profiling OpenMP programs

As in serial optimization, profiling tools can often hint at the root causes for performance problems also with OpenMP. In the simplest case, one could employ any of the methods described in Section 2.1 on a per-thread basis and compare the different scalar profiles. This strategy has several drawbacks, the most important being that scalar tools have no concept of specific OpenMP features. In a scalar profile, OpenMP constructs like team forks, barriers, spin loops, locks, critical sections, and even parts of user code that were packed into a separate function by the compiler appear as normal functions whose purpose can only be deduced from some more or less cryptic name.

More advanced tools allow for direct determination of load imbalance, serial fraction, OpenMP loop overhead, etc. (see below for more discussion regarding those issues). At the time of writing, very few production-grade free tools are available for OpenMP profiling, and the introduction of tasking in the OpenMP 3.0 standard has complicated matters for tool developers.

Figure 7.1 shows an *event timeline* comparison between two runs of the same code, using Intel's Thread Profiler for Windows [T23]. An event timeline contains information about the behavior of the application over time. In the case of OpenMP profiling, this pertains to typical constructs like parallel loops, barriers, locks, and also performance issues like load imbalance or insufficient parallelism. As a simple benchmark we choose the multiplication of a lower triangular matrix with a vector:

```
1    do k=1,NITER
2  !$OMP PARALLEL DO SCHEDULE(RUNTIME)
3      do row=1,N
4        do col=1,row
5          C(row) = C(row) + A(col,row) * B(col)
6        enddo
7      enddo
8  !$OMP END PARALLEL DO
9    enddo
```

(Note that privatizing the inner loop variable is not required here because this is automatic in Fortran, but not in C/C++.) If static scheduling is used, this problem obviously suffers from severe load imbalance. The bottom panel in Figure 7.1 shows the timeline for two threads and `STATIC` scheduling. The lower thread (shown in black) is a "shepherd" thread that exists for administrative purposes and can be ignored because it does not execute user code. Until the first parallel region is encountered, only one thread executes. After that, each thread is shown using different colors or shadings which encode the kind of activity it performs. Hatched areas denote "spinning," i.e., the thread waits in a loop until given more work to do or until it hits a barrier (actually, after some time, spinning threads are often put to sleep so as to free resources; this can be observed here just before the second barrier. This behavior can usually be influenced by the user). As expected, the static schedule leads to strong load imbalance so that more than half of the first thread's CPU time is wasted. With `STATIC,16` scheduling (top panel), the imbalance is gone and performance is improved by about 30%.

Thread profilers are usually capable of much more than just timeline displays. Often, a simple overall summary denoting, e.g., the fraction of walltime spent in barriers, spin loops, critical sections, or locks can reveal the nature of some performance problem.

7.2 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 discussed in Chapter 5. Communication (in terms of data transfer) is usually not much of an issue on shared memory as the access latencies inside a compute node are small and bandwidths are large (see, however, Chapter 8 for problems connected to memory access on ccNUMA architectures). The load imbalance problem can often be solved by choosing a suitable OpenMP scheduling strategy (see Section 6.1.6). However there are also very specific performance problems that are inherently connected to shared-memory programming in general and OpenMP in particular. In this section we will try to give some practical advice for avoiding typical OpenMP performance traps.

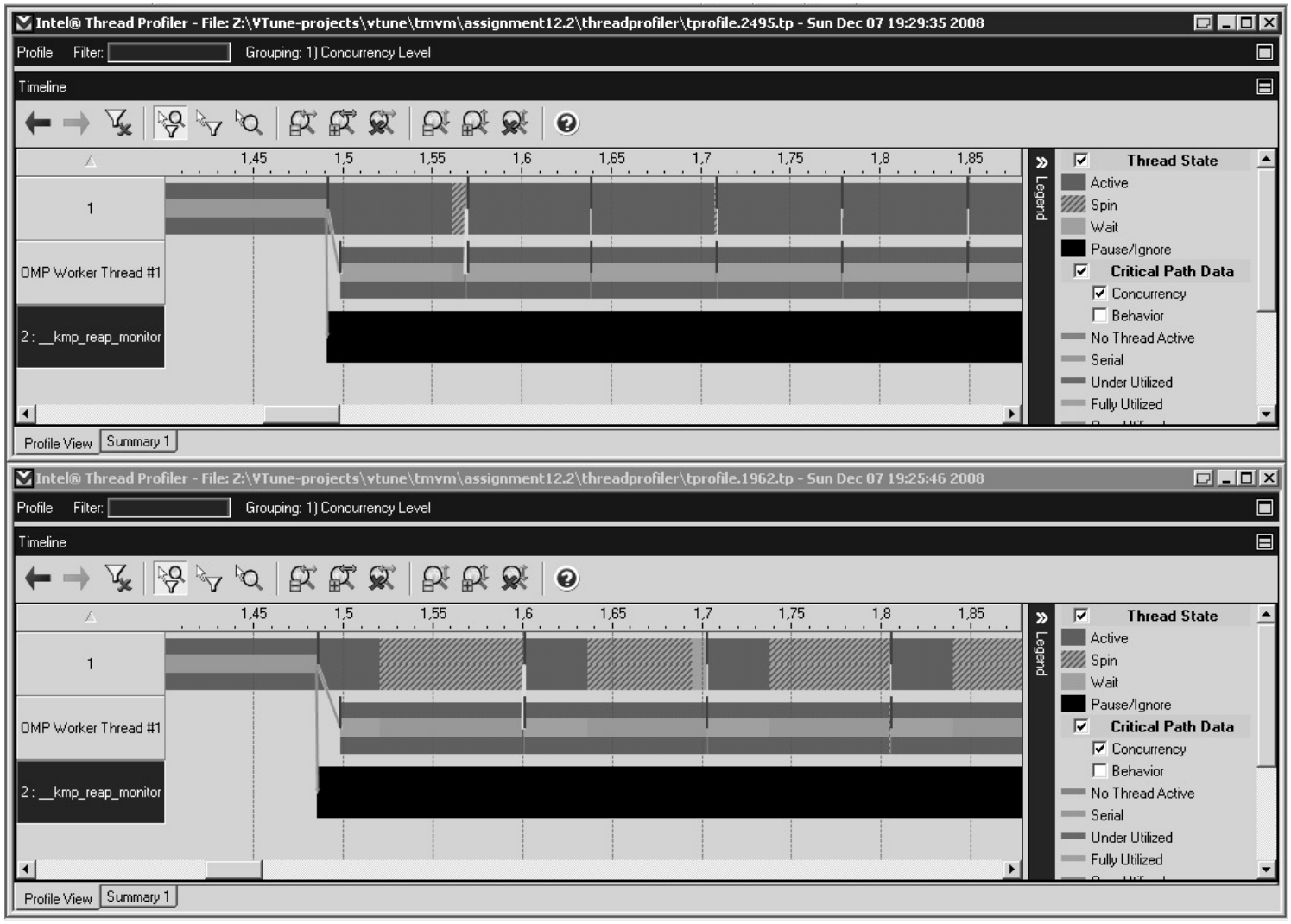

Figure 7.1: (See color insert after page 262.) Event timeline comparison of a threaded code (triangular matrix-vector multiplication) with `STATIC,16` (top panel) and `STATIC` (bottom panel) OpenMP scheduling.

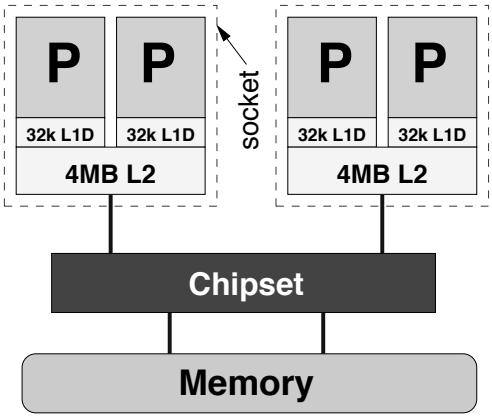

Figure 7.2: Dual-socket dual-core Xeon 5160 node used for most benchmarks in this chapter.

7.2.1 Ameliorating the impact of OpenMP worksharing constructs

Whenever a parallel region is started or stopped or a parallel loop is initiated or ended, there is some nonnegligible 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, in the case of tasking or dynamic/guided scheduling schemes each thread that becomes available must be supplied with a new task to work on, and the default barrier at the end of worksharing constructs or parallel regions synchronizes all threads. In terms of the refined scalability models discussed in Section 5.3.6, these contributions could be counted as "communication overhead." Since they tend to be linear in the number of threads, it seems that OpenMP is not really suited for strong scaling scenarios; with N threads, the speedup is

$$S_{\mathrm{omp}}(N) = \frac{1}{s+(1-s)/N+\kappa N+\lambda} \ , \tag{7.1}$$

with λ denoting N-independent overhead. For large N this expression goes to zero, which seems to be a definite showstopper for the OpenMP programming model. In practice, however, all is not lost: The performance impact depends on the actual values of κ and λ, of course. If some simple guidelines are followed, the adverse effects of OpenMP overhead can be much reduced:

Run serial code if parallelism does not pay off

This is perhaps the most popular performance issue with OpenMP. If the worksharing construct does not contain enough "work" per thread because, e.g., each iteration of a short loop executes in a short time, OpenMP overhead will lead to very bad performance. It is then better to execute a serial version if the loop count is below some threshold. The OpenMP `IF` clause helps with this:

```
1 !$OMP PARALLEL DO IF(N>1700)
2   do i=1,N
3     A(i) = B(i) + C(i) * D(i)
4   enddo
5 !$OMP END PARALLEL DO
```

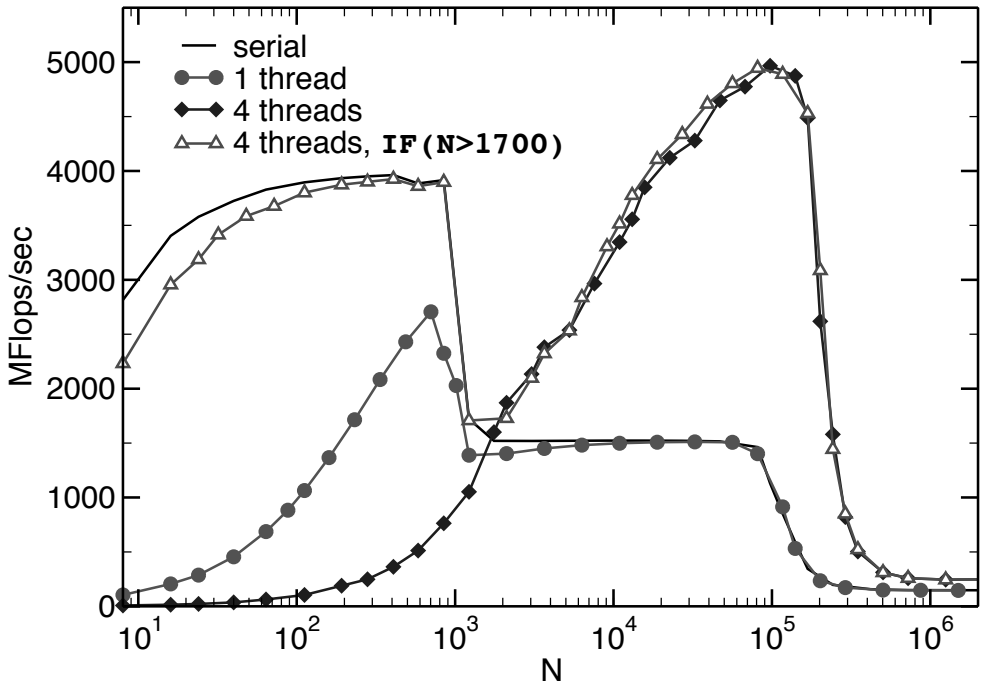

Figure 7.3: OpenMP overhead and the benefits of the `IF(N>1700)` clause for the vector triad benchmark. (Dual-socket dual-core Intel Xeon 5160 3.0 GHz system like in Figure 7.2, Intel compiler 10.1).

Figure 7.3 shows a comparison of vector triad data in the purely serial case and with one and four OpenMP threads, respectively, on a dual-socket Xeon 5160 node (sketched in Figure 7.2). The presence of OpenMP causes overhead at small N even if only a single thread is used (see below for more discussion regarding the cost of worksharing constructs). Using the `IF` clause leads to an optimal combination of threaded and serial loop versions if the threshold is chosen appropriately, and is hence mandatory when large loop lengths cannot be guaranteed. However, at $N \lesssim 1000$ there is still some measurable performance hit; after all, more code must be executed than in the purely serial case. Note that the `IF` clause is a (partial) cure for the symptoms, but not the reasons for parallel loop overhead. The following sections will elaborate on methods that can actually reduce it.

Instead of disabling parallel execution altogether, it may also be an option to reduce the number of threads used on a particular parallel region by means of the `NUM_THREADS` clause:

```
1 !$OMP PARALLEL DO NUM_THREADS(2)
2   do i=1,N
3     A(i) = B(i) + C(i) * D(i)
4   enddo
5 !$OMP END PARALLEL DO
```

Fewer threads mean less overhead, and the resulting performance may be better than with `IF`, at least for some loop lengths.

Avoid implicit barriers

Be aware that most OpenMP worksharing constructs (including `OMP DO`/`END DO`) insert automatic barriers at the end. This is the safe default, 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. Implicit barriers add to synchronization overhead like critical regions, but they are often required to protect from race conditions. The programmer should check carefully whether the `NOWAIT` clause is really safe.

Section 7.2.2 below will show how barrier overhead for the standard case of a worksharing loop can be determined experimentally.

Try to minimize the number of parallel regions

This is often formulated as the need to parallelize loop nests on a level as far out as possible, and it is inherently connected to the previous guidelines. Parallelizing inner loop levels leads to increased OpenMP overhead because a team of threads is spawned or woken up multiple times:

```
  double precision :: R
  R = 0.d0
  do j=1,N
!$OMP PARALLEL DO REDUCTION(+:R)
    do i=1,N
      R = R + A(i,j) * B(i)
    enddo
!$OMP END PARALLEL DO
    C(j) = C(j) + R
  enddo
```

In this particular example, the team of threads is restarted N times, once in each iteration of the `j` loop. Pulling the complete parallel construct to the outer loop reduces the number of restarts to one and has the additional benefit that the `reduction` clause becomes obsolete as all threads work on disjoint parts of the result vector:

```
!$OMP PARALLEL DO
  do j=1,N
    do i=1,N
```

```
4         C(j) = C(j) + A(i,j) * B(i)
5       enddo
6     enddo
7   !$OMP END PARALLEL DO
```

The less often the team of threads needs to be forked or restarted, the less overhead is incurred. This strategy may require some extra programming care because if the team continues running between worksharing constructs, code which would otherwise be executed by a single thread will be run by all threads redundantly. Consider the following example:

```
1     double precision :: R,S
2     R = 0.d0
3   !$OMP PARALLEL DO REDUCTION(+:R)
4     do i=1,N
5       A(i) = DO_WORK(B(i))
6       R = R + B(i)
7     enddo
8   !$OMP END PARALLEL DO
9     S = SIN(R)
10  !$OMP PARALLEL DO
11    do i=1,N
12      A(i) = A(i) + S
13    enddo
14  !$OMP END PARALLEL DO
```

The `SIN` function call between the loops is performed by the master thread only. At the end of the first loop, all threads synchronize and are possibly even put to sleep, and they are started again when the second loop executes. In order to circumvent this overhead, a continuous parallel region can be employed:

```
1     double precision :: R,S
2     R = 0.d0
3   !$OMP PARALLEL PRIVATE(S)
4   !$OMP DO REDUCTION(+:R)
5     do i=1,N
6       A(i) = DO_WORK(B(i))
7       R = R + B(i)
8     enddo
9   !$OMP END DO
10    S = SIN(R)
11  !$OMP DO
12    do i=1,N
13      A(i) = A(i) + S
14    enddo
15  !$OMP END DO NOWAIT
16  !$OMP END PARALLEL
```

Now the `SIN` function in line 10 is computed by all threads, and consequently `S` must be privatized. It is safe to use the `NOWAIT` clause on the second loop in order to reduce barrier overhead. This is not possible with the first loop as the final result of the reduction will only be valid after synchronization.

Avoid "trivial" load imbalance

The number of tasks, or the parallel loop trip count, should be large compared to the number of threads. If the trip count is a small noninteger multiple of the number of threads, some threads will end up doing significantly less work than others, leading to load imbalance. This effect is independent of any other load balancing or overhead issues, i.e., it occurs even if each task comprises exactly the same amount of work, and also if OpenMP overhead is negligible.

A typical situation where it may become important is the execution of deep loop nests on highly threaded architectures [O65] (see Section 1.5 for more information on hardware threading). The larger the number of threads, the fewer tasks per thread are available on the parallel (outer) loop:

```
1   double precision, dimension(N,N,N,M) :: A
2 !$OMP PARALLEL DO SCHEDULE(STATIC) REDUCTION(+:res)
3   do l=1,M
4     do k=1,N
5       do j=1,N
6         do i=1,N
7           res = res + A(i,j,k,l)
8         enddo ; enddo ; enddo ; enddo
9 !$OMP END PARALLEL DO
```

The outer loop is the natural candidate for parallelization here, causing the minimal number of executed worksharing loops (and implicit barriers) and generating the least overhead. However, the outer loop length `M` may be quite small. Under the best possible conditions, if t is the number of threads, $t - \mathrm{mod}(\mathtt{M},t)$ threads receive a chunk that is one iteration smaller than for the other threads. If $\mathtt{M}/t$ is small, load imbalance will hurt scalability.

The `COLLAPSE` clause (introduced with OpenMP 3.0) can help here. For *perfect loop nests*, i.e., with no code between the different `do` (and `enddo`) statements and loop counts not depending on each other, the clause collapses a specified number of loop levels into one. Computing the original loop indices is taken care of automatically, so that the loop body can be left unchanged:

```
1   double precision, dimension(N,N,N,M) :: A
2 !$OMP PARALLEL DO SCHEDULE(STATIC) REDUCTION(+:res) COLLAPSE(2)
3   do l=1,M
4     do k=1,N
5       do j=1,N
6         do i=1,N
7           res = res + A(i,j,k,l)
8   enddo ; enddo ; enddo ; enddo
9 !$OMP END PARALLEL DO
```

Here the outer two loop levels are collapsed into one with a loop length of $\mathtt{M}\times\mathtt{N}$, and the resulting long loop will be executed by all threads in parallel. This ameliorates the load balancing problem.

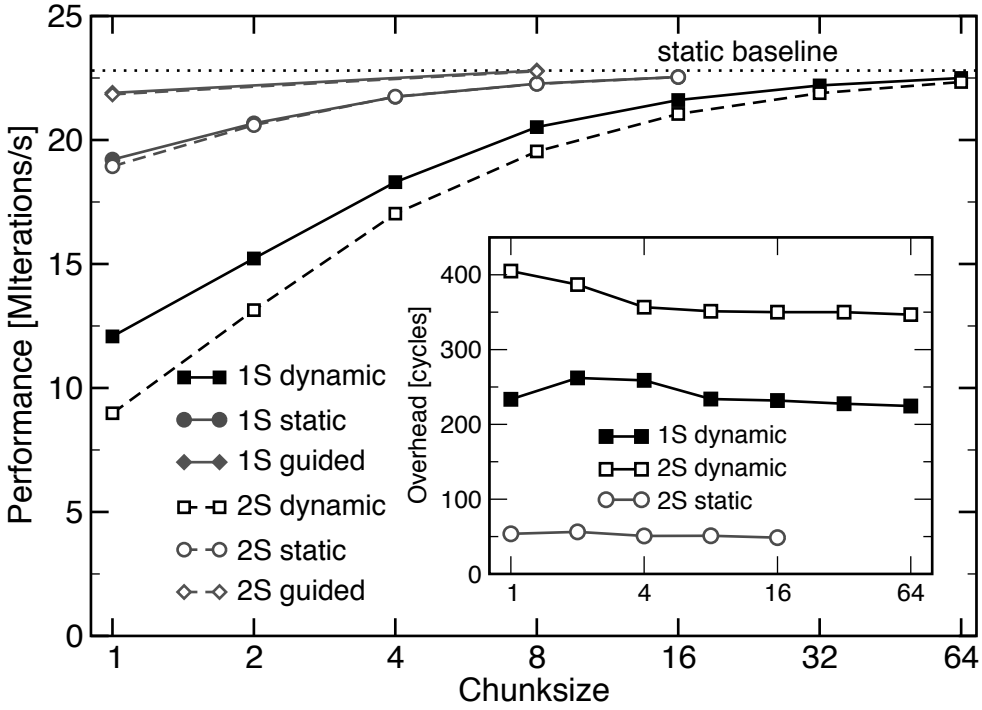

Figure 7.4: Main panel: Performance of a trivial worksharing loop with a large loop count under static (filled circles) versus dynamic (open symbols) scheduling on two cores in an L2 group (1S) or on different sockets (2S) of a dual-socket dual-core Intel Xeon 5160 3.0 GHz system like in Figure 7.2 (Intel Compiler 10.1). Inset: Overhead in processor cycles for assigning a single chunk to a thread.

Avoid dynamic/guided loop scheduling or tasking unless necessary

All parallel loop scheduling options (except `STATIC`) and tasking constructs require some amount of nontrivial computation or bookkeeping in order to figure out which thread is to compute the next chunk or task. This overhead can be significant if each task contains only a small amount of work. One can get a rough estimate for the cost of assigning a new loop chunk to a thread with a simple benchmark test. Figure 7.4 shows a performance comparison between static and dynamic scheduling for a simple parallel loop with purely computational workload:

```
1 !$OMP PARALLEL DO SCHEDULE(RUNTIME) REDUCTION(+:s)
2   do i=1,N
3     s = s + compute(i)
4   enddo
5 !$OMP END PARALLEL DO
```

The `compute()` function performs some in-register calculations (like, e.g., transcendental function evaluations) for some hundreds of cycles. It is unimportant what it does exactly, as long as it neither interferes with the main parallel loop nor incurs bandwidth bottlenecks on any memory level. `N` should be chosen large enough so that

OpenMP loop startup overhead becomes negligible. The performance baseline with t threads, $P_s(t)$, is then measured with static scheduling without a chunksize, in units of million iterations per second (see the dotted line in Figure 7.4 for two threads on two cores of the Xeon 5160 dual-core dual-socket node depicted in Figure 7.2). This baseline does not depend on the binding of the threads to cores (inside cache groups, across ccNUMA domains, etc.) because each thread executes only a single chunk, and any OpenMP overhead occurs only at the start and at the end of the worksharing construct. At large `N`, the static baseline should thus be t times larger than the purely serial performance.

Whenever a chunksize is used with any scheduling variant, assigning a new chunk to a thread will take some time, which counts as overhead. The main panel in Figure 7.4 shows performance data for static (circles), dynamic (squares), and guided (diamonds) scheduling when the threads run in an L2 cache group (closed symbols) or on different sockets (open symbols), respectively. As expected, the overhead is largest for small chunks, and dominant only for dynamic scheduling. Guided scheduling performs best because larger chunks are assigned at the beginning of the loop, and the indicated chunksize is just a lower bound (see Figure 6.2). The difference between intersocket and intra-L2 situations is only significant with dynamic scheduling, because some common resource is required to arbitrate work distribution. If this resource can be kept in a shared cache, chunk assignment will be much faster. It will also be faster with guided scheduling, but due to the large *average* chunksize, the effect is unnoticeable.

If $P(t,c)$ is the t-thread performance at chunksize c, the difference in per-iteration per-thread execution times between the static baseline and the "chunked" version is the per-iteration overhead. Per complete chunk, this is

$$T_o(t,c) = \frac{t}{c}\left(\frac{1}{P(t,c)} - \frac{1}{P_s(t)}\right) . \tag{7.2}$$

The inset in Figure 7.4 shows that the overhead in CPU cycles is roughly independent of chunksize, at least for chunks larger than 4. Assigning a new chunk to a thread costs over 200 cycles if both threads run inside an L2 group, and 350 cycles when running on different sockets (these times include the 50 cycle penalty per chunk for static scheduling). Again we encounter a situation where mutual thread placement, or affinity, is decisive.

Please note that, although the general methodology is applicable to all shared-memory systems, the results of this analysis depend on hardware properties and the actual implementation of OpenMP worksharing done by the compiler. The actual numbers may differ significantly on other platforms.

Other factors not recognized by this benchmark can impact the performance of dynamically scheduled worksharing constructs. In memory-bound loop code, prefetching may be inefficient or impossible if the chunksize is small. Moreover, due to the indeterministic nature of memory accesses, ccNUMA locality could be hard to maintain, which pertains to guided scheduling as well. See Section 8.3.1 for details on this problem.

7.2.2 Determining OpenMP overhead for short loops

The question arises how one can estimate the possible overheads that go along with a parallel worksharing construct. In general, the overhead consists of a constant part and a part that depends on the number of threads. There are vast differences from system to system as to how large it can be, but it is usually of the order of at least hundreds if not thousands of CPU cycles. It can be determined easily by fitting a simple performance model to data obtained from a low-level benchmark. As an example we pick the vector triad with short vector lengths and static scheduling so that the parallel run scales across threads if each core has its own L1 (performance would not scale with larger vectors as shared caches or main memory usually present bottlenecks, especially on multicore systems):

```
1 !$OMP PARALLEL PRIVATE(j)
2   do j=1,NITER
3 !$OMP DO SCHEDULE(static) NOWAIT   ! nowait is optional (see text)
4     do i=1,N
5       A(i) = B(i) + C(i) * D(i)
6     enddo
7 !$OMP END DO
8   enddo
9 !$OMP END PARALLEL
```

As usual, `NITER` is chosen so that overall runtime can be accurately measured and one-time startup effects (loading data into cache the first time etc.) become unimportant. The `NOWAIT` clause is optional and is used here only to demonstrate the impact of the implied barrier at the end of the loop worksharing construct (see below).

The performance model assumes that overall runtime with a problem size of N on t threads can be split into computational and setup/shutdown contributions:

$$T(N,t) = T_\mathrm{c}(N,t) + T_\mathrm{s}(t) \; . \tag{7.3}$$

Further assuming that we have measured purely serial performance $P_\mathrm{s}(N)$ we can write

$$T_\mathrm{c}(N,t) = \frac{2N}{tP_\mathrm{s}(N/t)} \; , \tag{7.4}$$

which allows an N-dependent performance behavior unconnected to OpenMP overhead. The factor of two in the numerator accounts for the fact that performance is measured in MFlops/sec and each loop iteration comprises two Flops. As mentioned above, setup/shutdown time is composed of a constant latency and a component that depends on the number of threads:

$$T_\mathrm{s}(t) = T_\mathrm{l} + T_\mathrm{p}(t) \; . \tag{7.5}$$

Now we can calculate parallel performance on t threads at problem size N:

$$P(N,t) = \frac{2N}{T(N,t)} = \frac{2N}{2N\left[tP_\mathrm{s}(N/t)\right]^{-1} + T_\mathrm{s}(t)} \tag{7.6}$$

Figure 7.5 shows performance data for the small-N vector triad on a node with four cores (two sockets), including parametric fits to the model (7.6) for the one- (circles) and four-thread (squares) cases, and with the implied barrier removed (open symbols) by a `nowait` clause. The indicated fit parameters in nanoseconds denote $T_s(t)$, as defined in (7.5). The values measured for P_s with the serial code version are not used directly for fitting but approximated by

$$P_s(N) = \frac{2N}{2N/P_{\max} + T_0} \ , \tag{7.7}$$

where $P_{\max}$ is the asymptotic serial triad performance and T_0 summarizes all the scalar overhead (pipeline startup, loop branch misprediction, etc.). Both parameters are determined by fitting to measured serial performance data (dotted-dashed line in Figure 7.5). Then, (7.7) is used in (7.6):

$$P(N,t) = \frac{1}{(tP_{\max})^{-1} + (T_0 + T_s(t))/2N} \tag{7.8}$$

Surprisingly, there is a measurable overhead for running with a single OpenMP thread versus purely serial mode. However, as the two versions reach the same asymptotic performance at $N \lesssim 1000$, this effect cannot be attributed to inefficient scalar code, although OpenMP does sometimes prevent advanced loop optimizations. The single-thread overhead originates from the inability of the compiler to generate a separate code version for serial execution.

The barrier is negligible with a single thread, and accounts for most of the overhead at four threads. But even without it about 190 ns (570 CPU cycles) are spent for setting up the worksharing loop in that case (one could indeed estimate the average memory latency from the barrier overhead, assuming that the barrier is implemented by a memory variable which must be updated by all threads in turn). The data labeled "restart" (diamonds) was obtained by using a combined `parallel do` directive, so that the team of threads is woken up each time the inner loop gets executed:

```
1   do j=1,NITER
2 !$OMP PARALLEL DO SCHEDULE(static)
3     do i=1,N
4       A(i) = B(i) + C(i) * D(i)
5     enddo
6 !$OMP END PARALLEL DO
7   enddo
```

This makes it possible to separate the thread wakeup time from the barrier and worksharing overheads: There is an additional cost of nearly 460 ns (1380 cycles) for restarting the team, which proves that it can indeed be beneficial to minimize the number of parallel regions in an OpenMP program, as mentioned above.

Similar to the experiments performed with dynamic scheduling in the previous section, the actual overhead incurred for small OpenMP loops depends on many factors like the compiler and runtime libraries, organization of caches, and the general node structure. It also makes a significant difference whether the thread team resides

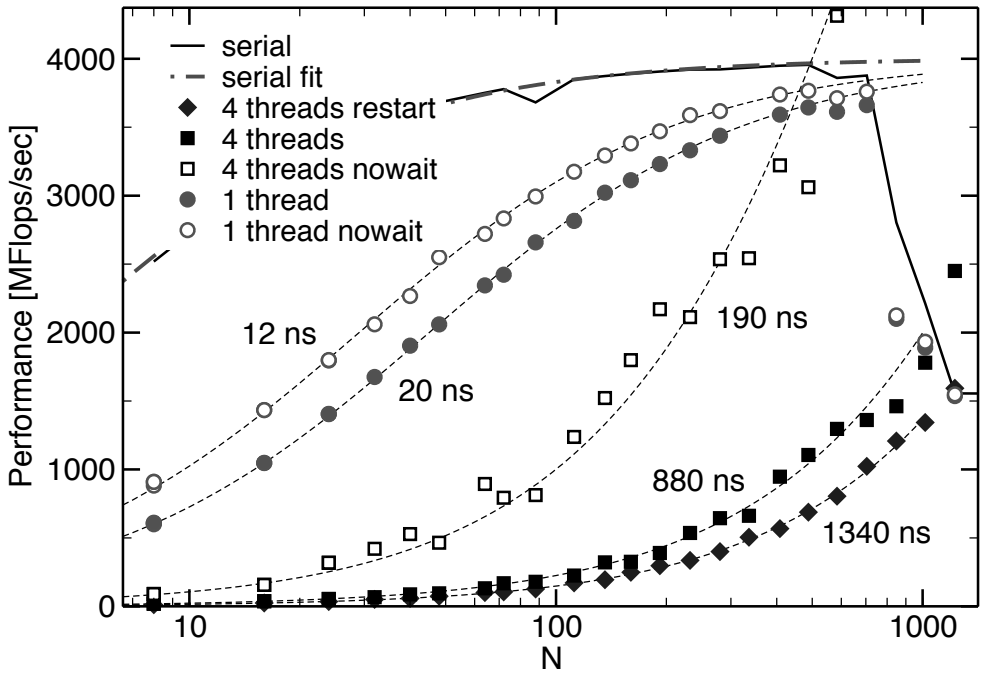

Figure 7.5: OpenMP loop overhead $T_s(t)$ for the small-N vector triad, determined from fitting the model (7.6) to measured performance data. Note the large impact of the implied barrier (removed by the `nowait` clause). The diamonds indicate data obtained by restarting the parallel region on every triad loop. (Intel Xeon 5160 3.0 GHz dual-core dual-socket node (see Figure 7.2), Intel Compiler 10.1.)

inside a cache group or encompasses several groups [M41] (see Appendix A for information on how to use affinity mechanisms). It is a general result, however, that implicit (and explicit) barriers are the largest contributors to OpenMP-specific parallel overhead. The EPCC OpenMP microbenchmark suite [136] provides a framework for measuring many OpenMP-related timings, including barriers, but use an approach slightly different from the one shown here.

7.2.3 Serialization

The most straightforward way to coordinate access to shared resources is to use critical region. Unless used with caution, these bear the potential of serializing the code. In the following example, columns of the matrix `M` are updated in parallel. Which columns get updated is determined by the `col_calc()` function:

```
1   double precision, dimension(N,N) :: M
2   integer :: c
3   ...
4  !$OMP PARALLEL DO PRIVATE(c)
5   do i=1,K
6     c = col_calc(i)
7  !$OMP CRITICAL
8     M(:,c) = M(:,c) + f(c)
```

```
9  !$OMP END CRITICAL
10   enddo
11 !$OMP END PARALLEL DO
```

Function `f()` returns an array, which is added to column `c` of the matrix. Since it is possible that a column is updated more than once, the array summation is protected by a critical region. However, if most of the CPU time is spent in line 8, the program is effectively serialized and there will be no gain from using more than one thread; the program will even run slower because of the additional overhead for entering a critical construct.

This coarse-grained way of protecting a resource (matrix `M` in this case) is often called a *big fat lock*. One solution could be to make use of the resource's substructure, i.e., the property of the matrix to consist of individual columns, and protect access to each column separately. Serialization can then only occur if two threads try to update the same column at the same time. Unfortunately, named critical regions (see Section 6.1.4) are of no use here as the name cannot be a variable. However, it is possible to use a separate OpenMP lock variable for each column:

```
1    double precision, dimension(N,N) :: M
2    integer (kind=omp_lock_kind), dimension(N) :: locks
3    integer :: c
4  !$OMP PARALLEL
5  !$OMP DO
6    do i=1,N
7      call omp_init_lock(locks(i))
8    enddo
9  !$OMP END DO
10   ...
11 !$OMP DO PRIVATE(c)
12   do i=1,K
13     c = col_calc(i)
14     call omp_set_lock(locks(c))
15     M(:,c) = M(:,c) + f(c)
16     call omp_unset_lock(locks(c))
17   enddo
18 !$OMP END DO
19 !$OMP END PARALLEL
```

If the mapping of `i` to `c`, mediated by the `col_calc()` function, is such that not only a few columns get updated, parallelism is greatly enhanced by this optimization (see [A83] for a real-world example). One should be aware though that setting and unsetting OpenMP locks incurs some overhead, as is the case for entering and leaving a critical region. If this overhead is comparable to the cost for updating a matrix row, the fine-grained synchronization scheme is of no use.

There is a solution to this problem if memory is not a scarce resource: One may use thread-local copies of otherwise shared data that get "pulled together" by, e.g., a reduction operation at the end of a parallel region. In our example this can be achieved most easily by doing an OpenMP reduction on `M`. If `K` is large enough, the additional cost is negligible:

```
  double precision, dimension(1:N,1,N) :: M
  integer :: c
  ...
!$OMP PARALLEL DO PRIVATE(c) REDUCTION(+:M)
  do i=1,K
    c = col_calc(i)
    M(:,c) = M(:,c) + f(c)
  enddo
!$OMP END PARALLEL DO
```

Note that reductions on arrays are only allowed in Fortran, and some further restrictions apply [P11]. In C/C++ it would be necessary to perform explicit privatization inside the parallel region (probably using heap memory) and do the reduction manually, as shown in Listing 6.2.

In general, privatization should be given priority over synchronization when possible. This may require a careful analysis of the costs involved in the needed copying and reduction operations.

7.2.4 False sharing

The hardware-based cache coherence mechanisms described in Section 4.2.1 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 in order to avoid serialization on the shared resource, array `S`:

```
  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
```

```
12      IND = A(i)
13      S(ID,IND) = S(ID,IND) + 1
14    enddo
15  !$OMP END DO NOWAIT
16    ! calculate complete histogram
17  !$OMP CRITICAL
18    do j=1,8
19      S(0,j) = S(0,j) + S(ID,j)
20    enddo
21  !$OMP END CRITICAL
22  !$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, corresponding to two or three cache lines on most processors. On each histogram update to `S` in line 13, the writing CPU must gain exclusive ownership of one of those cache lines; almost every write leads to a cache miss and subsequent coherence traffic because it is likely that the needed cache line resides in another processor's cache, in modified state. Compared to 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 13.

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, allocating `S` in line 6 as `S(0:NT*CL,8)`, with `CL` being the cache line size in integers, will reserve an exclusive cache line for each thread. Of course, the first index to `S` will have to be multiplied by `CL` everywhere else in the program (transposing `S` will save some memory, but the main principle stays the same).

An even more painless solution exists in the form of data privatization (see also Section 7.2.3 above): 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 to the purely serial version by using the `REDUCTION` clause:

```
1   integer, dimension(8) :: S
2   integer :: IND
3   S=0
4  !$OMP PARALLEL DO PRIVATE(IND) REDUCTION(+:S)
5   do i=1,N
6     IND = A(i)
7     S(IND) = S(IND) + 1
```

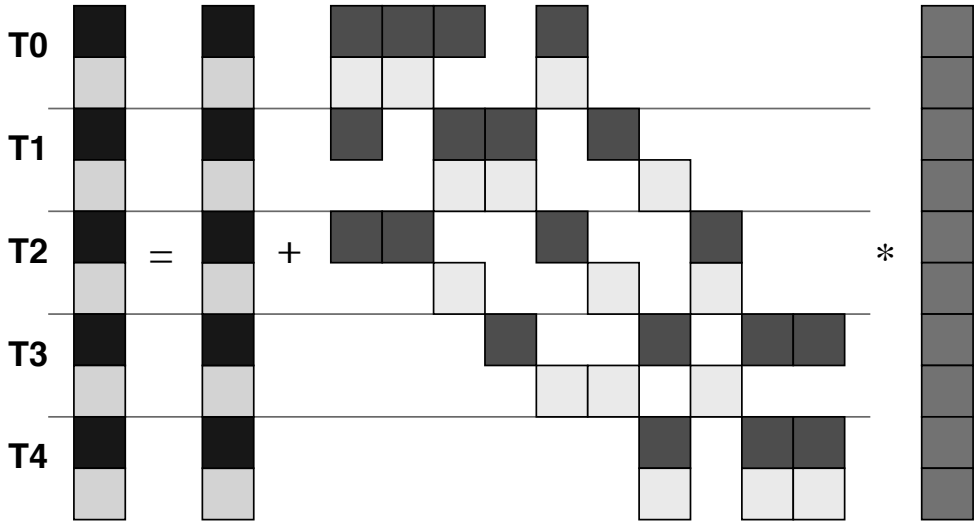

Figure 7.6: Parallelization approach for sparse MVM (five threads). All marked elements are handled in a single iteration of the parallelized loop. The RHS vector is accessed by all threads.

```
8     enddo
9   !$OMP EMD PARALLEL DO
```

Setting `S` to zero is only required if the code is to be compiled without OpenMP support, as the reduction clause automatically initializes the variables in question with appropriate starting values.

Again, privatization in its most convenient form (reduction) is possible here because we are using Fortran (the OpenMP standard allows no reductions on arrays in C/C++) and the elementary operation (addition) is supported for the `REDUCTION` clause. However, even without the clause the required operations are easy to formulate explicitly (see Problem 7.1).

7.3 Case study: Parallel sparse matrix-vector multiply

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 Section 3.6 [A84, A82].

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 Figure 7.6). For the CRS matrix format, this principle can be applied in a straightforward manner:

```
1   !$OMP PARALLEL DO PRIVATE(j)[1]
2     do i = 1,N_r
```

[1]The privatization of inner loop indices in the lexical extent of a parallel outer loop is not strictly required in Fortran, but it is in C/C++ [P11].

```
3     do j = row_ptr(i), row_ptr(i+1) - 1
4       C(i) = C(i) + val(j) * B(col_idx(j))
5     enddo
6   enddo
7  !$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 load 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:

```
1  !$OMP PARALLEL PRIVATE(diag,diagLen,offset)
2   do diag=1, N_j
3     diagLen = jd_ptr(diag+1) - jd_ptr(diag)
4     offset = jd_ptr(diag) - 1
5  !$OMP DO
6     do i=1, diagLen
7       C(i) = C(i) + val(offset+i) * B(col_idx(offset+i))
8     enddo
9  !$OMP END DO
10  enddo
11 !$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 Figure 3.19), the outer loop over all blocks is a natural candidate:

```
1  !$OMP PARALLEL DO PRIVATE(block_start,block_end,i,diag,
2  !$OMP& diagLen,offset)
3   do ib=1,N_r,b
4     block_start = ib
5     block_end   = min(ib+b-1,N_r)
6     do diag=1,N_j
7       diagLen = jd_ptr(diag+1)-jd_ptr(diag)
8       offset  = jd_ptr(diag) - 1
9       if(diagLen .ge. block_start) then
10        do i=block_start, min(block_end,diagLen)
11          C(i) = C(i)+val(offset+i)*B(col_idx(offset+i))
12        enddo
13      endif
14    enddo
15  enddo
16 !$OMP END PARALLEL DO
```

This version has even got less OpenMP overhead because the `DO` directive is on the outermost loop. Unfortunately, there is even more potential for load imbalance because of the matrix rows being sorted for size. But as the dependence of workload

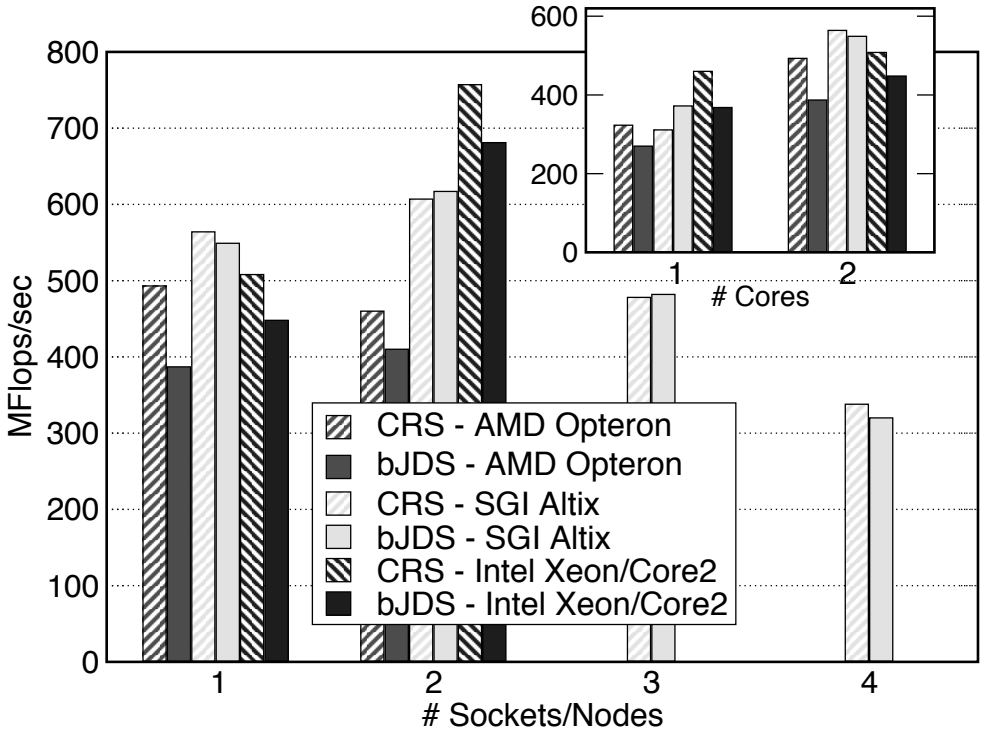

Figure 7.7: Performance and strong scaling for straightforward OpenMP parallelization of sparse MVM on three different architectures, comparing CRS (hatched bars) and blocked JDS (solid bars) variants. The Intel Xeon/Core2 system is of UMA type, the other two systems are ccNUMA. The different scaling baselines have been separated (one socket/LD in the main frame, one core in the inset).

on loop index is roughly predictable, a static schedule with a chunksize of one can remedy most of this effect.

Figure 7.7 shows performance and scaling behavior of the parallel CRS and blocked JDS versions on three different architectures: Two ccNUMA systems (Opteron and SGI Altix, equivalent to the block diagrams in Figures 4.5 and 4.6), and one UMA system (Xeon/Core2 node like in Figure 4.4). In all cases, the code was run on as few locality domains or sockets as possible, i.e., first filling one LD or socket before going to the next. The inset displays intra-LD or intrasocket scalability with respect to the single-core scaling baseline. All systems considered are strongly bandwidth-limited on this level. The performance gain from using a second thread is usually far from a factor of two, as may be expected. Note, however, that this behavior also depends crucially on the ability of one core to utilize the local memory interface: The relatively low single-thread CRS performance on the Altix leads to a significant speedup of approximately 1.8 for two threads (see also Section 5.3.8).

Scalability across sockets or LDs (main frame in Figure 7.7) reveals a crucial difference between ccNUMA and UMA systems. Only the UMA node shows the expected speedup when the second socket gets used, due to the additional bandwidth provided by the second frontside bus (it is a known problem of FSB-based designs that bandwidth scalability across sockets is less than perfect, so we don't see a fac-

tor of two here). Although ccNUMA architectures should be able to deliver scalable bandwidth, both code versions seem to be extremely unsuitable for ccNUMA, exhibiting poor scalability or, in case of the Altix, even performance breakdowns at larger numbers of LDs.

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. See Chapter 8 for programming techniques that can mitigate this problem, and [O66] for a more general assessment of parallel sparse MVM optimization on modern shared-memory systems.

Problems

For solutions see page 301 *ff*.

7.1 *Privatization gymnastics*. In Section 7.2.4 we have optimized a code for parallel histogram calculation by eliminating false sharing. The final code employs a `REDUCTION` clause to sum up all the partial results for `S()`. How would the code look like in C or C++?

7.2 *Superlinear speedup*. When the number of threads is increased at constant problem size (strong scaling), making additional cache space available to the application, there is a chance that the whole working set fits into the aggregated cache of all used cores. The speedup will then be larger than what the additional number of cores seem to allow. Can you identify this situation in the performance data for the 2D parallel Jacobi solver (Figure 6.3)? Of course, this result may be valid only for one special type of machine. What condition must hold for a general cache-based machine so that there is at least a chance to see superlinear speedup with this code?

7.3 *Reductions and initial values*. In some of the examples for decreasing the number of parallel regions on page 170 we have explicitly set the reduction variable `R` to zero before entering the parallel region, although OpenMP initializes such variables on its own. Why is it necessary then to do it anyway?

7.4 *Optimal thread count*. What is the optimal thread count to use for a memory-bound multithreaded application on a two-socket ccNUMA system with six cores per socket and a three-core L3 group?