Chapter 8

Locality optimizations on ccNUMA architectures

It was mentioned already in the section on ccNUMA architecture that, for applications whose performance is bound by memory bandwidth, locality and contention problems (see Figures 8.1 and 8.2) 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 (3.0) does not refer to page placement at all and it is up to the programmer to use the tools that system builders provide. This chapter discusses the general, i.e., mostly system-independent options for correct data placement, and possible pitfalls that may prevent it. We will also show that page placement is not an issue that is restricted to shared-memory parallel programming.

8.1 Locality of access on ccNUMA

Although ccNUMA architectures are ubiquitous today, the need for ccNUMA-awareness has not yet arrived in all application areas; memory-bound code must be designed to employ proper page placement [O67]. 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

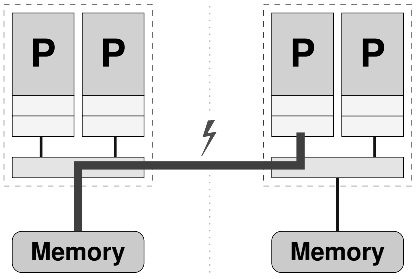

Figure 8.1: Locality problem on a ccNUMA system. Memory pages got mapped into a locality domain that is not connected to the accessing processor, leading to NUMA traffic.

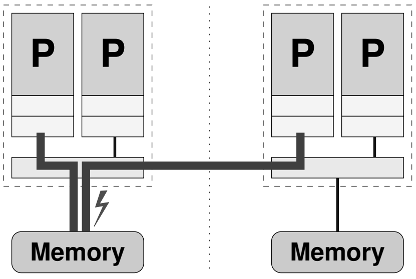

Figure 8.2: Contention problem on a ccNUMA system. Even if the network is very fast, a single locality domain can usually not saturate the bandwidth demands from concurrent local and nonlocal accesses.

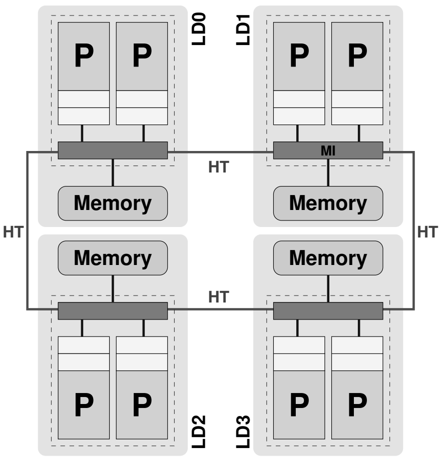

Figure 8.3: A ccNUMA system (based on dual-core AMD Opteron processors) with four locality domains LD0... LD3 and two cores per LD, coupled via a HyperTransport network. There are three NUMA access levels (local domain, one hop, two hops).

across the network. In this context, "mapping" means that a page table entry is set up, which describes the association of a physical with a virtual memory page. Consequently, locality of access in ccNUMA systems is always followed on the OS 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, although the problem tends to be cache-bound in this case anyway. 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 the following we assume that appropriate affinity mechanisms have been employed (see Appendix A).

A typical ccNUMA node with four locality domains is depicted in Figure 8.3. It uses two HyperTransport (HT) links per socket to connect to neighboring domains, which results in a "closed chain" topology. Memory access is hence categorized into three levels, depending on how many HT hops (zero, one, or two) are required to reach the desired page. The actual remote bandwidth and latency penalties can vary significantly across different systems; vector triad measurements can at least provide rough guidelines. See the following sections for details about how to control page placement.

Note that even with an extremely fast NUMA interconnect whose bandwidth and latency are comparable to local memory access, the contention problem cannot be eliminated. No interconnect, no matter how fast, can turn ccNUMA into UMA.

8.1.1 Page placement by first touch

Fortunately, the initial mapping requirement can be fulfilled in a portable manner on all current ccNUMA architectures. If configured correctly (this pertains to firmware ["BIOS"], operating system and runtime libraries alike), they support a *first touch* policy for memory pages: A page gets mapped into the locality domain of the processor that first writes to it. Merely *allocating* memory is not sufficient.

It is therefore the data initialization code that deserves attention on ccNUMA (and using `calloc()` in C will most probably be counterproductive). As an example we look again at a naïve OpenMP-parallel implementation of the vector triad code from Listing 1.1. Instead of allocating arrays on the stack, however, we now use dynamic (heap) memory for reasons which will be explained later (we omit the timing functionality for brevity):

```
1    double precision, allocatable, dimension(:) :: A, B, C, D
2    allocate(A(N), B(N), C(N), D(N))
3  ! initialization
4    do i=1,N
5      B(i) = i; C(i) = mod(i,5); D(i) = mod(i,10)
6    enddo
7    ...
8    do j=1,R
9  !$OMP PARALLEL DO
10     do i=1,N
11       A(i) = B(i) + C(i) * D(i)
12     enddo
13 !$OMP END PARALLEL DO
14     call dummy(A,B,C,D)
15   enddo
```

Here we have explicitly written out the loop which initializes arrays `B`, `C`, and `D` with sensible data (it is not required to initialize `A` because it will not be read before being written later). If this code, which is prototypical for many OpenMP applications that have not been optimized for ccNUMA, is run across several locality domains, it will not scale beyond the maximum performance achievable on a single LD if the working set does not fit into cache. This is because the initialization loop is executed by a single thread, writing to `B`, `C`, and `D` for the first time. Hence, all memory pages belonging to those arrays will be mapped into a single LD. As Figure 8.4 shows, the consequences are significant: If the working set fits into the aggregated cache, scalability is good. For large arrays, however, 8-thread performance (filled circles) drops even below the 2-thread (one LD) value (open diamonds), because all threads access memory in LD0 via the HT network, leading to severe contention. As mentioned above, this problem can be solved by performing array initialization in parallel. The loop from lines 4–6 in the code above should thus be replaced by:

```
! initialization
!$OMP PARALLEL DO
  do i=1,N
    B(i) = i; C(i) = mod(i,5); D(i) = mod(i,10)
  enddo
!$OMP END PARALLEL DO
```

This simple modification, which is actually a no-op on UMA systems, makes a huge difference on ccNUMA in memory-bound situations (see open circles and inset in Figure 8.4). Of course, in the very large N limit where the working set does not fit into a single locality domain, data will be "automatically" distributed, but not

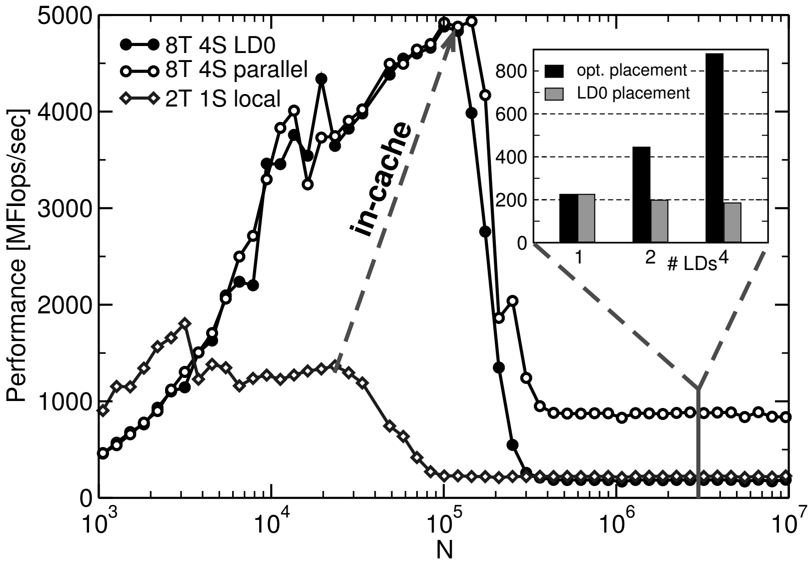

Figure 8.4: Vector triad performance and scalability on a four-LD ccNUMA machine like in Figure 8.3 (HP DL585 G5), comparing data for 8 threads with page placement in LD0 (filled circles) with correct parallel first touch (open circles). Performance data for local access in a single LD is shown for reference (open diamonds). Two threads per socket were used throughout. In-cache scalability is unharmed by unsuitable page placement. For memory-bound situations, putting all data into a single LD has ruinous consequences (see inset).

in a controlled way. This effect is by no means something to rely on when data distribution is key.

Sometimes it is not sufficient to just parallelize array initialization, for instance if there is no loop to parallelize. In the OpenMP code on the left of Figure 8.5, initialization of `A` is done in a serial region using the `READ` statement in line 8. The access to `A` in the parallel loop will then lead to contention. 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, data will be distributed across the locality domains. Array `B` does not have to be initialized but will automatically be mapped correctly.

There are some requirements that must be fulfilled for first-touch to work properly and result in good loop performance scalability:

- The OpenMP loop schedules of initialization and work loops must obviously be identical and reproducible, i.e., the only possible choice is `STATIC` with a constant chunksize, and the use of tasking is ruled out. Since the OpenMP standard does not define a default schedule, it is 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. The only simple option is the choice of

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




! executed on single LD
READ(1000) A
! contention 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)
!$OMP PARALLEL DO
  do i=1,N
    A(i) = 0.d0
  enddo
!$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
```

Figure 8.5: Optimization by correct NUMA placement. Left: The `READ` statement is executed by a single thread, placing `A` to a single locality domain. Right: Doing parallel initialization leads to correct distribution of pages across domains.

an appropriate chunksize (as small as possible, but at least several pages of data). See Section 8.3.1 for more information about dynamic scheduling under ccNUMA conditions.

- For successive parallel loops with the same number of iterations and the same number of parallel threads, each thread should get the same part of the iteration space in both loops. The OpenMP 3.0 standard guarantees this behavior only if both loops use the `STATIC` schedule with the same chunksize (or none at all) and if they bind to the same parallel region. Although the latter condition can usually not be satisfied, at least not for all loops in a program, current compilers generate code which makes sure that the iteration space of loops of the same length and OpenMP schedule is always divided in the same way, even in different parallel regions.
- The hardware must actually be *capable* of scaling memory bandwidth across locality domains. This may not always be the case, e.g., if cache coherence traffic produces contention on the NUMA network.

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 [O68]. A discussion of some of the problems that emerge from the combination of OpenMP with C++ can be found in Section 8.4, and in [C100] and [C101].

It is not specified in a portable way how a page that has been allocated and initialized can lose its page table entry. In most cases it is sufficient to deallocate memory if it resides on the heap (using `DEALLOCATE` in Fortran, `free()` in C, or `delete[]` in C++). This is why we have reverted to the use of dynamic memory for the triad benchmarks described above. If a new memory block is allocated later on, the first touch policy will apply as usual. Even so, some optimized implementations of runtime libraries will not actually deallocate memory on `free()` but add the pages to a "pool" to be re-allocated later with very little overhead. In case of doubt, the system documentation should be consulted for ways to change this behavior.

Locality problems tend to show up most prominently with shared-memory parallel code. Independently running processes automatically employ first touch placement if each process keeps its affinity to the locality domain where it had initialized its data. See, however, Section 8.3.2 for effects that may yet impede strictly local access.

8.1.2 Access locality by other means

Apart from plain first-touch initialization, operating systems often feature advanced tools for explicit page placement and diagnostics. These facilities are highly nonportable by nature. Often there are command-line tools or configurable dynamic objects that influence allocation and first-touch behavior without the need to change source code. Typical capabilities include:

- Setting policies or preferences to restrict mapping of memory pages to specific locality domains, irrespective of where the allocating process or thread is running.
- Setting policies for distributing the mapping of successively touched pages across locality domains in a "round-robin" or even random fashion. If a shared-memory parallel program has erratic access patterns (e.g., due to limitations imposed by the need for load balancing), and a coherent first-touch mapping cannot be employed, this may be a way to get at least a limited level of parallel scalability for memory-bound codes. See also Section 8.3.1 for relevant examples.
- Diagnosing the current distribution of pages over locality domains, probably on a per-process basis.

Apart from stand-alone tools, there is always a library with a documented API, which provides more fine-grained control over page placement. Under the Linux OS, the `numatools` package contains all the functionality described, and also allows thread/process affinity control (i.e, to determine which thread/process should run where). See Appendix A for more information.

8.2 Case study: ccNUMA optimization of sparse MVM

It is now clear that the bad scalability of OpenMP-parallelized sparse MVM codes on ccNUMA systems (see Figure 7.7) is caused by contention due to the memory pages of the code's working set being mapped into a single locality domain on

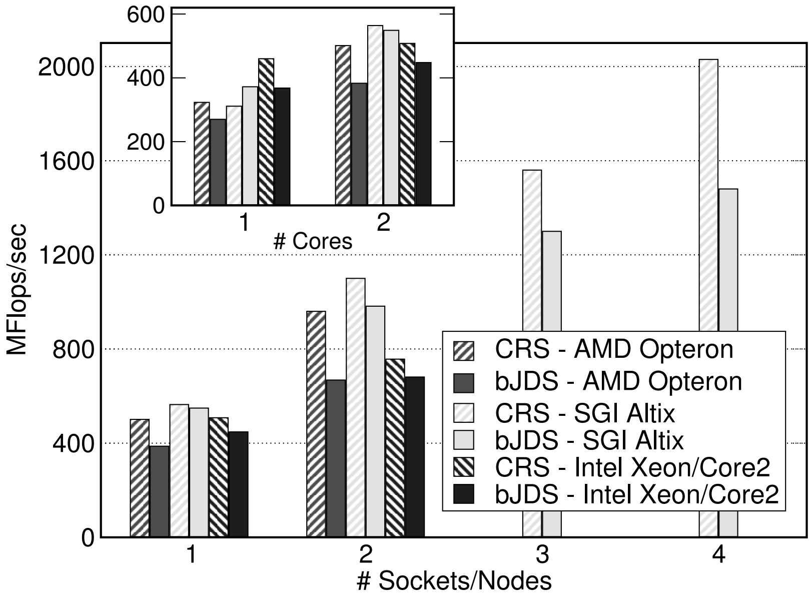

Figure 8.6: Performance and strong scaling for ccNUMA-optimized OpenMP parallelization of sparse MVM on three different architectures, comparing CRS (hatched bars) and blocked JDS (solid bars) variants. Cf. Figure 7.7 for performance without proper placement. The different scaling baselines have been separated (one socket/LD in the main frame, one core in the inset).

initialization. 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:

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

The initialization of `B` is based on the assumption that the nonzeros 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 RHS vector at all.

Figure 8.6 shows performance data for the same architectures and sMVM codes as in Figure 7.7 but with appropriate ccNUMA placement. There is no change in scalability for the UMA platform, which was to be expected, but also on the ccNUMA systems for up to two threads (see inset). 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 (the scaling baseline in the main panel is the locality domain or socket, respectively). 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 RHS vector into the correct locality domains, and thus leads to increased NUMA traffic.

8.3 Placement pitfalls

We have demonstrated that data placement is of premier importance on ccNUMA architectures, including commonly used two-socket cluster nodes. In principle, ccNUMA offers superior scalability for memory-bound codes, but UMA systems are much easier to handle and require no code optimization for locality of access. One can expect, though, that ccNUMA designs will prevail in the commodity HPC market, where dual-socket configurations occupy a price vs. performance "sweet spot." It must be emphasized, however, that the placement optimizations introduced in Section 8.1 may not always be applicable, e.g., when dynamic scheduling is unavoidable (see Section 8.3.1). Moreover, one may have arrived at the conclusion that placement problems are restricted to shared-memory programming; this is entirely untrue and Section 8.3.2 will offer some more insight.

8.3.1 NUMA-unfriendly OpenMP scheduling

As explained in Sections 6.1.3 and 6.1.7, dynamic/guided loop scheduling and OpenMP `task` constructs could be preferable over static work distribution in poorly load-balanced situations, if the additional overhead caused by frequently assigning tasks to threads is negligible. On the other hand, any sort of dynamic scheduling (including tasking) will necessarily lead to scalability problems if the thread team is spread across several locality domains. After all, the assignment of tasks to threads is unpredictable and even changes from run to run, which rules out an "optimal" page placement strategy.

Dropping parallel first touch altogether in such a situation is no solution as performance will then be limited by a single memory interface again. In order to get at least a significant fraction of the maximum achievable bandwidth, it may be best to distribute the working set's memory pages round-robin across the domains and hope for a statistically even distribution of accesses. Again, the vector triad can serve as a

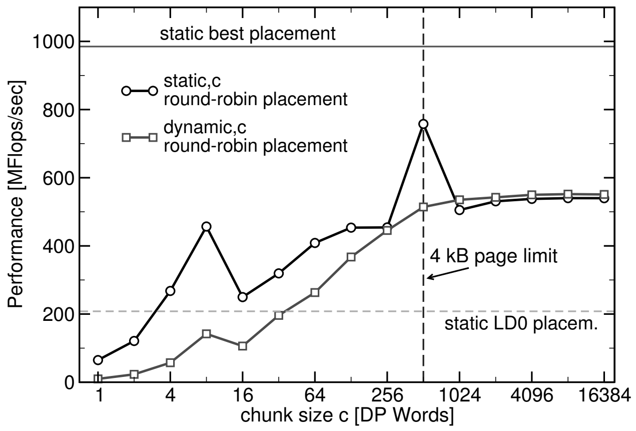

Figure 8.7: Vector triad performance vs. loop chunksize for static and dynamic scheduling with eight threads on a four-LD ccNUMA system (see Figure 8.3). Page placement was done round-robin on purpose. Performance for best parallel placement and LD0 placement with static scheduling is shown for reference.

convenient tool to fathom the impact of random page access. We modify the initialization loop by forcing static scheduling with a page-wide chunksize (assuming 4 kB pages):

```
! initialization
!$OMP PARALLEL DO SCHEDULE(STATIC,512)
  do i=1,N
    A(i) = 0; B(i) = i; C(i) = mod(i,5); D(i) = mod(i,10)
  enddo
!$OMP END PARALLEL DO
  ...
  do j=1,R
!$OMP PARALLEL DO SCHEDULE(RUNTIME)
    do i=1,N
      A(i) = B(i) + C(i) * D(i)
    enddo
!$OMP END PARALLEL DO
    call dummy(A,B,C,D)
  enddo
```

By setting the `OMP_SCHEDULE` environment variable, different loop schedulings can be tested. Figure 8.7 shows parallel triad performance versus chunksize c for static and dynamic scheduling, respectively, using eight threads on the four-socket ccNUMA system from Figure8.3. At large c, where a single chunk spans several memory pages, performance converges asymptotically to a level governed by random access across all LDs, independent of the type of scheduling used. In this case, 75% of all pages a thread needs reside in remote domains. Although this kind of erratic pattern bears at least a certain level of parallelism (compared with purely serial initialization as shown with the dashed line), there is almost a 50% performance penalty versus the ideal case (solid line). The situation at $c = 512$ deserves some attention: With static scheduling, the access pattern of the triad loop matches the placement policy from the initialization loop, enabling (mostly) local access in each LD. The residual discrepancy to the best possible result can be attributed to the ar-

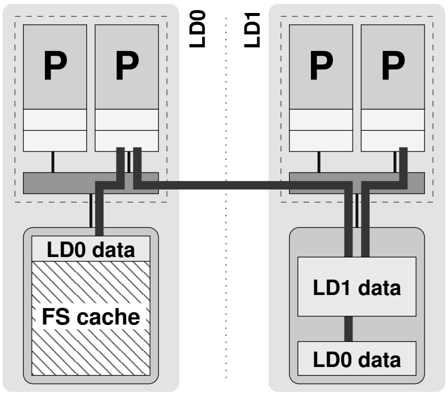

Figure 8.8: File system buffer cache can prevent locally touched pages to be placed in the local domain, leading to nonlocal access and contention. This is shown here for locality domain 0, where FS cache uses the major part of local memory. Some of the pages allocated and initialized by a core in LD0 get mapped into LD1.

rays not being aligned to page boundaries, which leads to some uncertainty regarding placement. Note that operating systems and compilers often provide means to align data structures to configurable boundaries (SIMD data type lengths, cache lines, and memory pages being typical candidates). Care should be taken, however, to avoid aliasing effects with strongly aligned data structures.

Although not directly related to NUMA effects, it is instructive to analyze the situation at smaller chunksizes as well. The additional overhead for dynamic scheduling causes a significant disadvantage compared to the static variant. If c is smaller than the cache line length (64 bytes here), each cache miss results in the transfer of a whole cache line of which only a fraction is needed, hence the peculiar behavior at $c \leq 64$. The interpretation of the breakdown at $c = 16$ and the gradual rise up until the page size is left as an exercise to the reader (see problems at the end of this chapter).

In summary, if purely static scheduling (without a chunksize) is ruled out, round-robin placement can at least exploit some parallelism. If possible, static scheduling with an appropriate chunksize should then be chosen for the OpenMP worksharing loops to prevent excessive scheduling overhead.

8.3.2 File system cache

Even if all precautions regarding affinity and page placement have been followed, it is still possible that scalability of OpenMP programs, but also overall system performance with independently running processes, is below expectations. Disk I/O operations cause operating systems to set up *buffer caches* which store recently read or written file data for efficient re-use. The size and placement of such caches is highly system-dependent and usually configurable, but the default setting is in most cases, although helpful for good I/O performance, less than fortunate in terms of ccNUMA locality.

See Figure 8.8 for an example: A thread or process running in LD0 writes a large file to disk, and the operating system reserves some space for a file system buffer cache in the memory attached to this domain. Later on, the same or another process in the this domain allocates and initializes memory ("LD0 data"), but not all of those

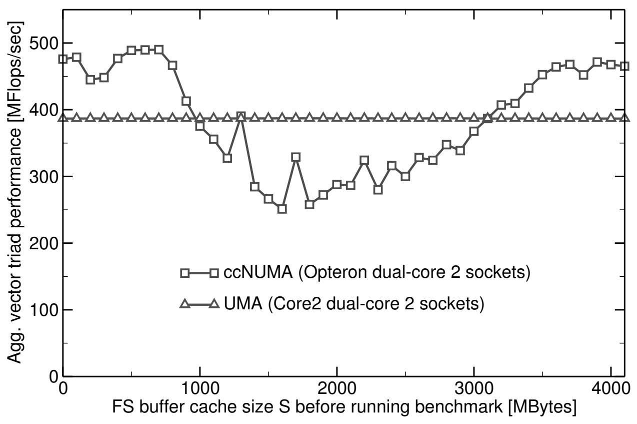

Figure 8.9: Performance impact of a large file system cache on a ccNUMA versus a UMA system (both with two sockets and four cores and 4 GB of RAM) when running four concurrent vector triads. The buffer cache was filled from a single core. See text for details. (Benchmark data by Michael Meier.)

pages fit into LD0 together with the buffer cache. By default, many systems then map the excess pages to another locality domain so that, even though first touch was correctly employed from the programmer's point of view, nonlocal access to LD0 data and contention at LD1's memory interface occurs.

A simple experiment can demonstrate this effect. We compare a UMA system (dual-core dual-socket Intel Xeon 5160 as in Figure 4.4) with a two-LD ccNUMA node (dual-core dual-socket AMD Opteron as in Figure 4.5), both equipped with 4 GB of main memory. On both systems we execute the following steps in a loop:

1. Write "dirty" data to disk and invalidate all the buffer cache. This step is highly system-dependent; usually there is a procedure that an administrator can execute[1] to do this, or a vendor-supplied library offers an option for it.

2. Write a file of size *S* to the local disk. The maximum file size equals the amount of memory. *S* should start at a very small size and be increased with every iteration of this loop until it equals the system's memory size.

3. Sync the cache so that there are no flush operations in the background (but the cache is still filled). This can usually be done by the standard UNIX `sync` command.

4. Run identical vector triad benchmarks on each core separately, using appropriate affinity mechanisms (see Appendix A). The aggregate size of all working sets should equal half of the node's memory. Report overall performance in MFlops/sec versus file size *S* (which equals the buffer cache size here).

The results are shown in Figure 8.9. On the UMA node, the buffer cache has certainly no impact at all as there is no concept of locality. On the other hand, the ccNUMA system shows a strong performance breakdown with growing file size and hits rock

[1]On a current Linux OS, this can be done by executing the command `echo 1 > /proc/sys/vm/drop_caches`. SGI Altix systems provide the `bcfree` command, which serves a similar purpose.

bottom when one LD is completely filled with buffer cache (at around 2 GB): This is when all the memory pages that were initialized by the triad loops in LD0 had to be mapped to LD1. Not surprisingly, if the file gets even larger, performance starts rising again because one locality domain is too small to hold even the buffer cache. If the file size equals the memory size (4 GB), parallel first touch tosses cache pages as needed and hence works as usual.

There are several lessons to be learned from this experiment. Most importantly it demonstrates that locality issues on ccNUMA are neither restricted to OpenMP (or generally, shared memory parallel) programs, nor is correct first touch a guarantee for getting "perfect" scalability. The buffer cache could even be a remnant from a previous job run by another user. Ideally there should be a way in production HPC environments to automatically "toss" the buffer cache whenever a production job finishes in order to leave a "clean" machine for the next user. As a last resort, if there are no user-level tools, and system administrators have not given this issue due attention, the normal user without special permissions can always execute a "sweeper" code, which allocates and initializes all memory:

```
  double precision, allocatable, dimension(:) :: A
  double precision :: tmp
  integer(kind=8) :: i
  integer(kind=8), parameter :: SIZE = SIZE_OF_MEMORY_IN_DOUBLES
  allocate A(SIZE)
  tmp=0.d0
! touch all pages
!$OMP PARALLEL DO
  do i=1, SIZE
    A(i) = SQRT(DBLE(i))  ! dummy values
  enddo
!$OMP END PARALLEL DO
! actually use the result
!$OMP PARALLEL DO
  do i=1,SIZE
    tmp = tmp + A(i)*A(1)
  enddo
!$OMP END PARALLEL DO
  print *,tmp
```

This code could also be used as part of a user application to toss buffer cache that was filled by I/O from the running program (this pertains to reading and writing alike). The second loop serves the sole purpose of preventing the compiler from optimizing away the first because it detects that A is never actually used. Parallelizing the loops is of course optional but can speed up the whole process. Note that, depending on the actual amount of memory and the number of "dirty" file cache blocks, this procedure could take a considerable amount of time: In the worst case, nearly all of main memory has to be written to disk.

Buffer cache and the resulting locality problems are one reason why performance results for parallel runs on clusters of ccNUMA nodes tend to show strong fluctuations. If many nodes are involved, a large buffer cache on only one of them can hamper the performance of the whole parallel application. It is the task of system administrators to exploit all options available for a given environment in order to lessen the impact of buffer cache. For instance, some systems allow to configure the strategy under which cache pages are kept, giving priority to local memory requirements and tossing buffer cache as needed.

8.4 ccNUMA issues with C++

Locality of memory access as shown above can often be implemented in languages like Fortran or C once the basic memory access patterns have been identified. Due to its object-oriented features, however, C++ is another matter entirely [C100, C101]. In this section we want to point out the most relevant pitfalls when using OpenMP-parallel C++ code on ccNUMA systems.

8.4.1 Arrays of objects

The most basic problem appears when allocating an array of objects of type `D` using the standard `new[]` operator. For simplicity, we choose `D` to be a simple wrapper around `double` with all the necessary overloaded operators to make it look and behave like the basic type:

```
class D {
  double d;
public:
  D(double _d=0.0) throw() : d(_d) {}
  ~D() throw() {}
  inline D& operator=(double _d) throw() {d=_d; return *this;}
  friend D operator+(const D&, const D&) throw();
  friend D operator*(const D&, const D&) throw();
  ...
};
```

Assuming correct implementation of all operators, the only difference between `D` and `double` should be that instantiation of an object of type `D` leads to immediate initialization, which is not the case for `doubles`, i.e., in `a=new D[N]`, memory allocation takes place as usual, but the default constructor gets called for each array member. Since `new` knows nothing about NUMA, these calls are all done by the thread executing `new`. As a consequence, all the data ends up in that thread's local memory. One way around this would be a default constructor that does not touch the member, but this is not always possible or desirable.

One should thus first map the memory pages that will be used for the array data to the correct nodes so that access becomes local for each thread, and then call the constructors to initialize the objects. This could be accomplished by *placement new*, where the number of objects to be constructed as well as the exact memory (base) address of their instantiation is specified. Placement `new` does not call any constructors, though. A simple way around the effort of using placement `new` is to overload

D::operator new[]. This operator has the sole responsibility to allocate "raw" memory. An overloaded version can, in addition to memory allocation, initialize the pages in parallel for good NUMA placement (we ignore the requirement to throw std::bad_alloc on failure):

```
void* D::operator new[](size_t n) {
  char *p = new char[n];       // allocate
  size_t i,j;
#pragma omp parallel for private(j) schedule(runtime)
  for(i=0; i<n; i += sizeof(D))
    for(j=0; j<sizeof(D); ++j)
      p[i+j] = 0;
  return p;
}

void D::operator delete[](void* p) throw() {
 delete [] static_cast<char*>p;
}
```

Construction of all objects in an array at their correct positions is then done automatically by the C++ runtime, using placement new. Note that the C++ runtime usually requests a little more space than would be needed by the aggregated object sizes, which is used for storing administrative information alongside the actual data. Since the amount of data is small compared to NUMA-relevant array sizes, there is no noticeable effect.

Overloading operator new[] works for simple cases like class D above. Dynamic members are problematic, however, because their NUMA locality cannot be easily controlled:

```
class E {
  size_t s;
  std::vector<double> *v;
public:
  E(size_t _s=100) : s(_s), v(new std::vector<double>(s)) {}
  ~E() { delete [] v; }
  ...
};
```

E's constructor initializes E::s and E::v, and these would be the only data items subject to NUMA placement by an overloaded E::operator new[] upon construction of an array of E. The memory addressed by E::v is not handled by this mechanism; in fact, the std::vector<double> is preset upon construction inside STL using copies of the object double(). This happens in the C++ runtime after E::operator new[] was executed. All the memory will be mapped into a single locality domain.

Avoiding this situation is hardly possible with standard C++ and STL constructs if one insists on constructing arrays of objects with new[]. The best advice is to call object constructors explicitly in a loop and to use a vector for holding pointers only:

```
  std::vector<E*> v_E(n);

```

```
#pragma omp parallel for schedule(runtime)
  for(size_t i=0; i<v_E.size(); ++i) {
    v_E[i] = new E(100);
  }
```

Since now the class constructor is called from different threads concurrently, it must be thread safe.

8.4.2 Standard Template Library

C-style array handling as shown in the previous section is certainly discouraged for C++; the STL `std::vector<>` container is much safer and more convenient, but has its own problems with ccNUMA page placement. Even for simple data types like `double`, which have a trivial default constructor, placement is problematic since, e.g., the allocated memory in a `std::vector<>(int)` object is filled with copies of `value_type()` using `std::uninitialized_fill()`. The design of a dedicated NUMA-aware container class would probably allow for more advanced optimizations, but STL defines a customizable abstraction layer called *allocators* that can effectively encapsulate the low-level details of a container's memory management. By using this facility, correct NUMA placement can be enforced in many cases for `std::vector<>` with minimal changes to an existing program code.

STL containers have an optional template argument by which one can specify the allocator class to use [C102, C103]. By default, this is `std::allocator<T>`. An allocator class provides, among others, the methods (class namespace omitted):

```
pointer allocate(size_type, const void *=0);
void deallocate(pointer, size_type);
```

Here `size_type` is `size_t`, and `pointer` is `T*`. The `allocate()` method gets called by the container's constructor to set up memory in much the same way as `operator new[]` for an array of objects. However, since all relevant supplementary information is stored in additional member variables, the number of bytes to allocate matches the space required by the container's contents only, at least on initial construction (see below). The second parameter to `allocate()` can supply additional information to the allocator, but its semantics are not standardized. `deallocate()` is responsible for freeing the allocated memory again.

The simplest NUMA-aware allocator would take care that `allocate()` not only allocates memory but initializes it in parallel. For reference, Listing 8.1 shows the code of a simple NUMA-friendly allocator, using standard `malloc()` for allocation. In line 19 the OpenMP API function `omp_in_parallel()` is used to determine whether the allocator was called from an active parallel region. If it was, the initialization loop is skipped. To use the template, it must be specified as the second template argument whenever a `std::vector<>` object is constructed:

```
 vector<double, NUMA_Allocator<double> > v(length);
```

Listing 8.1: A NUMA allocator template. The implementation is somewhat simplified from the requirements in the C++ standard.

```
template <class T> class NUMA_Allocator {
public:
  typedef T* pointer;
  typedef const T* const_pointer;
  typedef T& reference;
  typedef const T& const_reference;
  typedef size_t size_type;
  typedef T value_type;

  NUMA_Allocator() { }
  NUMA_Allocator(const NUMA_Allocator& _r) { }
  ~NUMA_Allocator() { }

  // allocate raw memory including page placement
  pointer allocate(size_type numObjects,
                   const void *localityHint=0) {
    size_type len = numObjects * sizeof(value_type);
    char *p = static_cast<char*>(std::malloc(len));
    if(!omp_in_parallel()) {
#pragma omp parallel for schedule(runtime) private(ofs)
      for(size_type i=0; i<len; i+=sizeof(value_type)) {
        for(size_type j=0; j<sizeof(value_type); ++j) {
          p[i+j]=0;
        }
    }
    return static_cast<pointer>(m);
  }

  // free raw memory
  void deallocate(pointer ptrToMemory, size_type numObjects) {
    std::free(ptrToMemory);
  }

  // construct object at given address
  void construct(pointer p, const value_type& x) {
    new(p) value_type(x);
  }

  // destroy object at given address
  void destroy(pointer p) {
    p-> value_type();
  }

private:
  void operator=(const NUMA_Allocator&) {}
};
```

What follows after memory allocation is pretty similar to the array-of-objects case, and has the same restrictions: The allocator's `construct()` method is called For each of the objects, and uses placement new to construct each object at the correct address (line 36). Upon destruction, each object's destructor is called explicitly (one of the rare cases where this is necessary) via the `destroy()` method in line 41. Note that container construction and destruction are not the only places where `construct()` and `destroy()` are invoked, and that there are many things which could destroy NUMA locality immediately. For instance, due to the concept of container size versus capacity, calling `std::vector<>::push_back()` just once on a "filled" container reallocates all memory plus a significant amount more, and copies the original objects to their new locations. The NUMA allocator will perform first-touch placement, but it will do so using the container's new capacity, not its size. As a consequence, placement will almost certainly be suboptimal. One should keep in mind that not all the functionality of `std::vector<>` is suitable to use in a ccNUMA environment. We are not even talking about the other STL containers (`deque`, `list`, `map`, `set`, etc.).

Incidentally, standard-compliant allocator objects of the same type must always compare as equal [C102]:

```
template <class T>
inline bool operator==(const NUMA_Allocator<T>&,
                       const NUMA_Allocator<T>&) { return true; }
template <class T>
inline bool operator!=(const NUMA_Allocator<T>&,
                       const NUMA_Allocator<T>&) { return false; }
```

This has the important consequence that an allocator object is necessarily stateless, ruling out some optimizations one may think of. A template specialization for `T=void` must also be provided (not shown here). These and other peculiarities are discussed in the literature. More sophisticated strategies than using plain `malloc()` do of course exist.

In summary we must add that the methods shown here are useful for outfitting existing C++ programs with *some* ccNUMA awareness without too much hassle. Certainly a newly designed code should be parallelized with ccNUMA in mind from the start.

Problems

For solutions see page 303 *ff.*

8.1 *Dynamic scheduling and ccNUMA.* When a memory-bound, OpenMP-parallel code runs on all sockets of a ccNUMA system, one should use static scheduling and initialize the data in parallel to make sure that memory accesses are mostly local. We want to analyze what happens if static scheduling is not a option, e.g., for load balancing reasons.

For a system with two locality domains, calculate the expected performance impact of dynamic scheduling on a memory-bound parallel loop. Assume for simplicity that there is exactly one thread (core) running per LD. This thread is able to saturate the local or any remote memory bus with some performance p. The inter-LD network should be infinitely fast, i.e., there is no penalty for non-local transfers and no contention effects on the inter-LD link. Further assume that all pages are homogeneously distributed throughout the system and that dynamic scheduling is purely statistical (i.e., each thread accesses all LDs in a random manner, with equal probability). Finally, assume that the chunksize is large enough so that there are no bad effects from hardware prefetching or partial cache line use.

The code's performance with static scheduling and perfect load balance would be $2p$. What is the expected performance under dynamic scheduling (also with perfect load balance)?

8.2 *Unfortunate chunksizes.* What could be possible reasons for the performance breakdown at chunksizes between 16 and 256 for the parallel vector triad on a four-LD ccNUMA machine (Figure 8.7)? Hint: Memory pages play a decisive role here.

8.3 *Speeding up "small" jobs.* If a ccNUMA system is sparsely utilized, e.g., if there are less threads than locality domains, and they all execute (memory-bound) code, is the first touch policy still the best strategy for page placement?

8.4 *Triangular matrix-vector multiplication.* Parallelize a triangular matrix-vector multiplication using OpenMP:

```
do r=1,N
  do c=1,r
    y(r) = y(r) + a(c,r) * x(c)
  enddo
enddo
```

What is the central parallel performance issue here? How can it be solved in general, and what special precautions are necessary on ccNUMA systems? You may ignore the standard scalar optimizations (unrolling, blocking).

8.5 *NUMA placement by overloading.* In Section 8.4.1 we enforced NUMA placement for arrays of objects of type `D` by overloading `D::operator new[]`. A similar thing was done in the NUMA-aware allocator class (Listing 8.1). Why did we use a loop nest for memory initialization instead of a single loop over `i`?