IN3200/IN4200: Chapter 8
Locality optimizations on ccNUMA architectures

Textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

Objective

- Utilize “maximum” memory bandwidth on ccNUMA architectures
    - Data placement
    - Locality of access

Recap of ccNUMA

- *Cache-coherent Nonuniform Memory Access* (ccNUMA) systems have a physically distributed memory that is *logically shared*. The aggregated memory appears as one single address space. Memory access performance depends on the which CPU (core) accesses which parts of memory (“local” vs. ”remote” access).
- Although there is a single address space (shared memory), there are private caches, or partially shared caches, for the different CPU cores (also true on UMA systems).
- *Cache coherence* protocols guarantee *consistency* between cached data and data in the shared memory at all times.

Locality domains

- A *locality domain* (LD) is a set of processor cores together with locally connected memory. This "local" memory can be accessed by the set of processor cores in the most efficient way, without resorting to a network of any kind.
- Each LD is a UMA building block.
- Multiple LDs are linked via a coherent interconnect, which can mediate direct, cache-coherent memory accesses. (This mechanism is transparent for the programmer.)

Example of ccNUMA

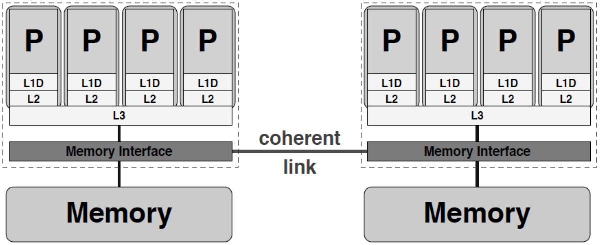


Figure 4.5: A ccNUMA system with two locality domains (one per socket) and eight cores.

Penalty for non-local transfers

The *locality problem*: Non-local memory transfers (between LDs) are more costly than local transfers (within a LD).

The *contention problem*: If two processors from different LDs access memory in the same LD, fighting for memory bandwidth.

Both problems can be “solved” (alleviated) by carefully observing the data access patterns of an application and restricting data access of each processor (mostly) to its own LD, through proper programming.

Locality and contention problems

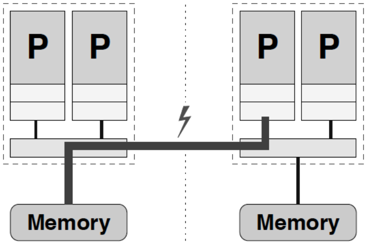


Figure 8.1: Locality problem on a cc-NUMA system. Memory pages got mapped into a locality domain that is not connected to the accessing processor, leading to NUMA traffic.

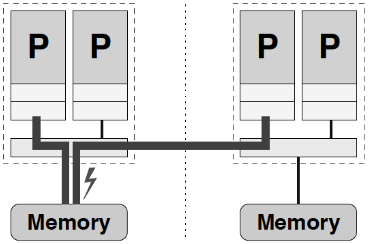


Figure 8.2: Contention problem on a cc-NUMA system. Even if the network is very fast, a single locality domain can usually not saturate the bandwidth demands from concurrent local and nonlocal accesses.

Both problems are due to “misplacement” of threads and data.

Example of non-uniform memory access costs

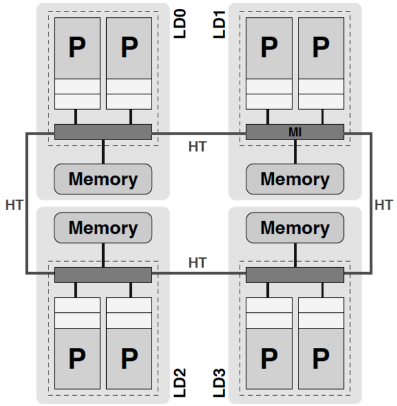


Figure 8.3: A ccNUMA system (based on dual-core AMD Opteron processors) with four locality domains LD0 … LD3 and two cores per LD, coupled via a HyperTransport network. There are three NUMA access levels (local domain, one hop, two hops).

The "placement" problem

Proper page placement is essential.

- Make sure that memory gets mapped into the locality domains of processors that actually access them. This minimizes NUMA traffic across the network. ("Mapping" means that a page table entry is set up, which describes the association of a physical with a virtual memory page.)
- Threads or processes must be pinned to those CPUs which had originally mapped their memory regions in order not to lose locality of access. (Need to use appropriate thread affinity mechanisms.)

Page placement by first touch

- Operating system and runtime libraries normally support a **first touch** policy for memory pages: A page gets mapped into the locality domain of the processor that first **writes** to it.
- **Note:** Merely *allocating* memory (such as `malloc`) is not sufficient.

Data initialization is key

- The data initialization part deserves close attention on ccNUMA.
    - Allocating arrays should use dynamic (heap) memory (`malloc`).
    - The `calloc` function will most probably be counterproductive.

A bad example

```
int *A, *B, *C, *D, i;

A = (int*)malloc(N*sizeof(int));
B = (int*)malloc(N*sizeof(int));
C = (int*)malloc(N*sizeof(int));
D = (int*)malloc(N*sizeof(int));

for (i=0; i<N; i++) {
  B[i] = i;
  C[i] = i%5;
  D[i] = i%10;
}

#pragma omp parallel for
for (i=0; i<N; i++)
  A[i] = B[i] + C[i]*D[i];
```

Why is the previous example bad?

- If the code 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.
- Problem: many "remote" accesses and severe contention

Fixing the initialization problem

```
int *A, *B, *C, *D, i;

A = (int*)malloc(N*sizeof(int));
B = (int*)malloc(N*sizeof(int));
C = (int*)malloc(N*sizeof(int));
D = (int*)malloc(N*sizeof(int));

#pragma omp parallel for
for (i=0; i<N; i++) {
  B[i] = i;
  C[i] = i%5;
  D[i] = i%10;
}
```

First touch by “fake” initialization

- Sometimes initializing an array with useful values can only be done by one thread.
- Then, a parallelized “fake” (unnecessary) initiation beforehand can secure the desirable first touch.

Some requirements

- The OpenMP loop schedules of initialization and work loops must be identical and reproducible.
  - Must use `static` scheduler and identical chunksize
- 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.
  - Also must use `static` scheduler and identical chunksize
- Beware of cache-coherence traffic, which can destroy the achievable aggregate memory bandwidth.
- Beware of global data (which is initialized before the `main()` function). Properly mapped local copies of global data may be a possible solution.

NUMA-unfriendly OpenMP scheduling

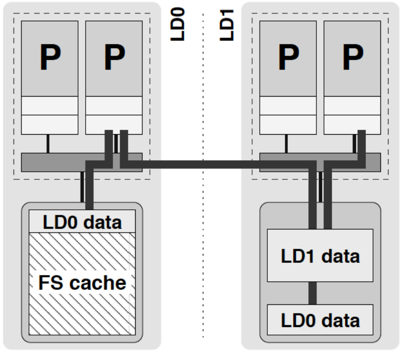
- Dynamic/guided loop scheduling and OpenMP task constructs could be preferable over static work distribution in poorly load-balanced situations.
- On the other hand, any sort of dynamic scheduling (including tasking) will necessarily lead to scalability problems if the thread team is spread across several locality domains.
- This is because the assignment of tasks to threads is unpredictable and even changes from run to run, which rules out an "optimal" page placement strategy.
- In such cases, it may be best to distribute the working set's memory pages round-robin (choosing a suitable chunksize) across the locality domains.

Another potential problem: File system cache

- 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 can be unfortunate with respect to ccNUMA locality.

An example of bad impact due to file system cache

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.


LD0
LD1
P
P
P
P
LD0 data
FS cache
LD1 data
LD0 data