CHAPTER 20

TBB on NUMA Architectures

Advanced programmers who care about performance know that exploiting locality is paramount. When it comes to locality, cache locality is the one that immediately springs to mind, but in many cases, for heavy-duty applications running on large shared-memory architectures, Non-Uniform Memory Access (NUMA) locality should also be considered. As you certainly know, NUMA conveys the message that memory is organized in different banks and some cores have faster access to some of the "close" banks than to "far" banks. More formally, a *NUMA node* is a grouping of the cores, caches, and local memory in which all cores share the same access time to the local shared caches and memory. Access time from one NUMA node to a different one can be significantly larger. Some questions arise, such as how the program data structures are allocated on the different NUMA nodes and where the threads that process these data structures are running (are they close or far from the data?). In this chapter, we address these questions, but more importantly, what can be done to exploit NUMA locality within a TBB parallel application.

Tuning for performance on NUMA systems comes down to four activities: (1) discovering what your platform topology is, (2) knowing the costs associated with accessing memory from the different nodes of your system, (3) controlling where your data is stored (data placement), and (4) controlling where your work executes (processor affinity).

In order to prevent you from being disappointed further down the line (i.e., to disappoint you right now!), we shall say the following upfront: currently, TBB does not offer high-level features for exploiting NUMA locality. Or in other words, out of the four activities listed before, TBB offers some help only in the fourth one, where we can rely on

M. Voss, R. Asenjo, J. Reinders, *Pro TBB*, https://doi.org/10.1007/978-1-4842-4398-5_20

the TBB `task_arena` (see Chapter 12) and local `task_sheduler_observer` (see Chapter 13) classes to identify the threads that should be confined in a NUMA node. For all the other activities, and even for the actual pinning of threads to NUMA nodes (which is the essential part of the fourth activity), we need to use either low-level OS-dependent system calls or higher-level third-party libraries and tools. This means, that even if this is a TBB book, this last chapter is not entirely about TBB. Our goal here is to thoroughly elaborate on how we can implement TBB code that exploits NUMA locality, even if most of the required activities are not directly related to TBB.

Now that we have warned the reader, let us break down the sections into which we have organized this chapter. We basically follow, in order, the four activities listed before. The first section shows some tools that can be used to discover the topology of our platform and to check how many NUMA nodes are available. If there is more than one NUMA node, we can move on to the next section. There, we use a benchmark to get an idea of the potential speedup that is at stake when exploiting NUMA locality on our particular platform. If the expected gain is convincing, we should start thinking in exploiting NUMA locality in our own code (not just in a simple benchmark). If we realize that our own problem can benefit from NUMA locality, we can jump into the heart of the matter which consists in mastering data placement and processor affinity. With this knowledge and with the help of TBB `task_arena` and `task_scheduler_observer` classes, we implement our first simple TBB application that exploits NUMA locality and assess the speedup obtained with respect to a baseline implementation. The whole process is summarized in Figure 20-1. We close the chapter sketching more advanced and general alternatives that could be considered for more complex applications.

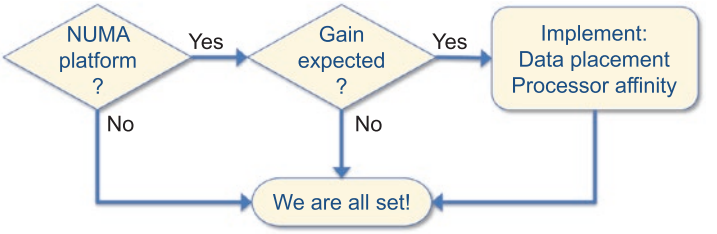

Figure 20-1. *Activities required to exploit NUMA locality*

Note If you are wondering why there is no high-level support in the current version of TBB, here are some reasons. First, it is a tough problem, highly dependent on the particular application that has to be parallelized and the architecture on which it will run. Since there is no one-size-fits-all solution, it is left to developers to determine the particular data placement and processor affinity alternatives that best suit the application at hand. Second, TBB architects and developers have always tried to avoid hardware specific solutions inside the TBB library because they can potentially hurt the portability of the code and the composability features of TBB. The library was not developed only to execute HPC applications, where we usually have exclusive access to the whole high-performance platform (or a partition of it). TBB should also do its best in shared environments in which other applications and processes are also running. Pinning threads to cores and memory to NUMA nodes can in many cases leads to suboptimal exploitation of the underlying architecture. Manually pinning has been shown repeatedly to be a bad idea in any application or system that has any dynamic nature in it at all. We strongly advise against taking such an approach, unless you are positive you will improve performance for your particular application on your particular parallel platform and you do not care about portability (or extra effort is made to implement a portable NUMA-aware application).

Considering the task-based nature of TBB parallel algorithms and the work-stealing scheduler that fuels the parallel execution, keeping the tasks running in cores close to the local memory can seem challenging. But that's not going to deter brave and fearless programmer like us. Let's go for it!

Discovering Your Platform Topology

"Know your enemy and yourself, and you shall win a hundred battles without loss." – Sun Tzu in *The Art of War*. This millenarian quote advises us to first strive to meticulously understand what we are facing before tackling it. There are some tools that come in handy to understand the underlying NUMA architecture. In this chapter, we will use `hwloc` and `likwid`[1] to gather information about the architecture and code execution. `hwloc` is a software package that

[1] `www.open-mpi.org/projects/hwloc` and `https://github.com/RRZE-HPC/likwid`.

provides a portable way to query information about the topology of a system, as well as to apply some NUMA controls, like data placement and processor affinity. `likwid` is another software package that informs about the hardware topology, can be used to collect hardware performance counters, and also provides a set of useful micro-benchmarks that can be used to characterize systems. We can also use VTune to analyze the performance of our code. Although `likwid` is only available for Linux, `hwloc` and VTune can be easily installed on Windows and MacOS as well. However, since the shared memory platforms that will serve to illustrate our codes run Linux, this will be the OS that we assume unless stated otherwise.

Because tuning for NUMA requires a deep understanding of the platforms being used, we will start by characterizing two machines that we will work on throughout this chapter. The two machines that we introduce next are known as **yuca** (from the yucca plant) and **aloe** (from the aloe vera plant). First, we can gather basic information about these machines. On Linux this information can be obtained using the command "`lscpu`", as we can see in Figure 20-2.

yuca

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                64
On-line CPU(s) list:   0-63
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             4
NUMA node(s):          4
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 46
Stepping:              6
CPU MHz:               1064.000
BogoMIPS:              3999.02
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              18432K
NUMA node0 CPU(s):     0-7,32-39
NUMA node1 CPU(s):     8-15,40-47
NUMA node2 CPU(s):     16-23,48-55
NUMA node3 CPU(s):     24-31,56-63
```

aloe

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    1
Core(s) per socket:    16
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Stepping:              2
CPU MHz:               1200.000
BogoMIPS:              4601.10
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              40960K
NUMA node0 CPU(s):     0-15
NUMA node1 CPU(s):     16-31
```

Figure 20-2. *Output of `lscpu` on yuca and aloe*

At first glance, we see that yuca has 64 logical cores numbered from 0 to 63, two logical cores per physical core (hyperthreading aka SMT or simultaneous multithreading, available), eight physical cores per socket, and four sockets that are also the four NUMA nodes or NUMA domains. For its part, aloe has 32 physical cores with hyperthreading disabled (only one thread per core), 16 physical cores per socket, and two sockets (NUMA nodes). At the end of the `lscpu` output, we can see the NUMA

nodes and the ids of the logical cores included in each node, but the picture will become clearer if we use the `lstopo` utility from the `hwloc` library. In Figure 20-3, we include the PDF file generated on yuca when executing the command `lstopo --no-io yuca.pdf` (the `--no-io` argument disregards the I/O device topology).

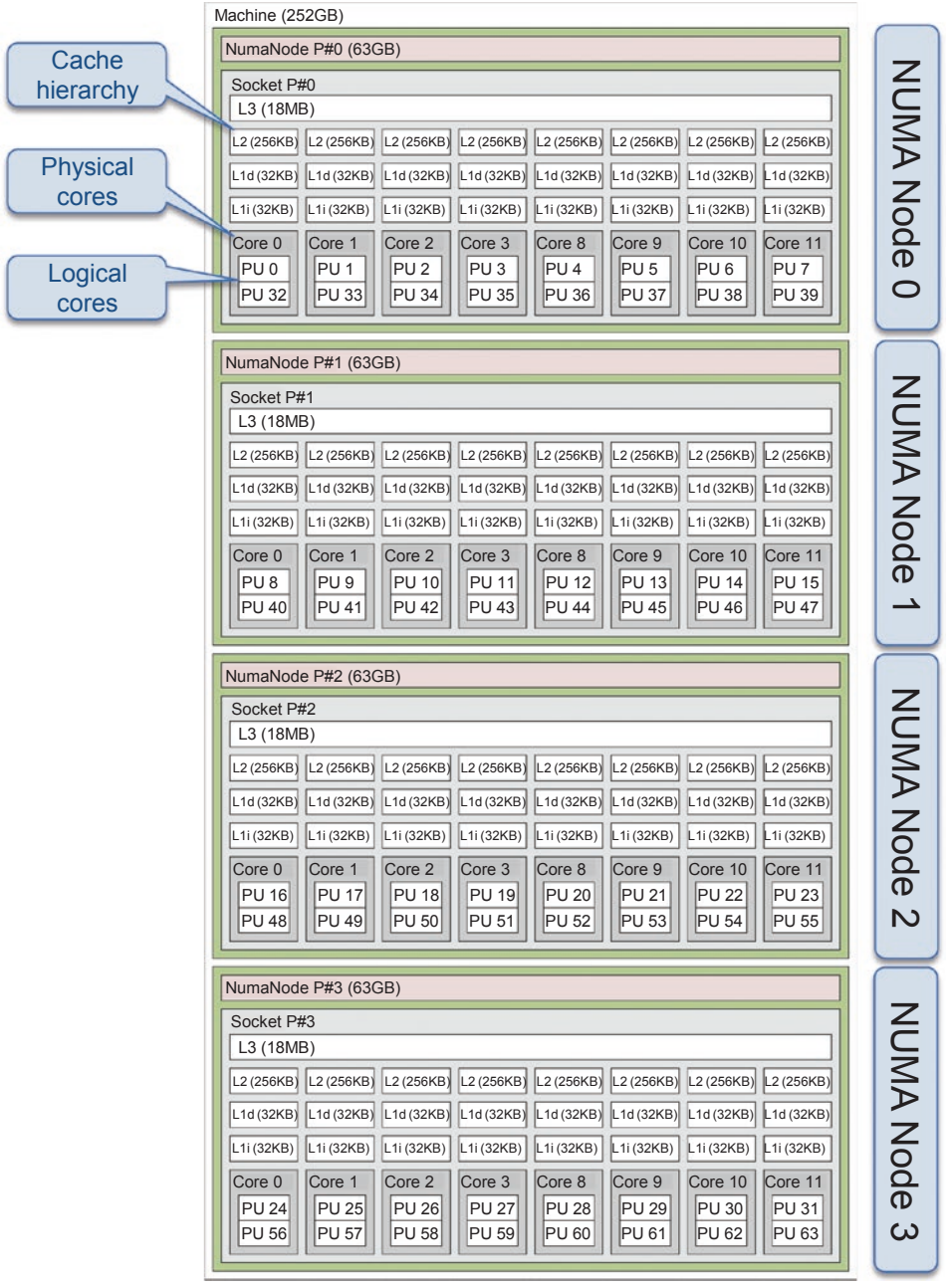

Figure 20-3. *Result of executing `lstopo` on yuca*

From this figure, we can get a clear representation of the NUMA organization in yuca. The four NUMA nodes include eight physical cores that are seen by the OS as 16 logical cores (also known as hardware threads). Note that logical core ids depend on the architecture, the firmware (BIOS configuration on PCs), and OS version, so we cannot assume anything from the numbering. For the particular configuration of yuca, logical cores 0 and 32 share the same physical core. Now we better understand the meaning of the last four lines of `lscpu` on yuca:

```
NUMA node0 CPU(s):     0-7,32-39
NUMA node1 CPU(s):     8-15,40-47
NUMA node2 CPU(s):     16-23,48-55
NUMA node3 CPU(s):     24-31,56-63
```

On yuca, each NUMA node has 63 GB of local memory, or 252 GB in total. Similarly, aloe also features 252 GB but organized in only two NUMA nodes. In Figure 20-4, we see a slightly edited version of the output of `lstopo` on aloe.

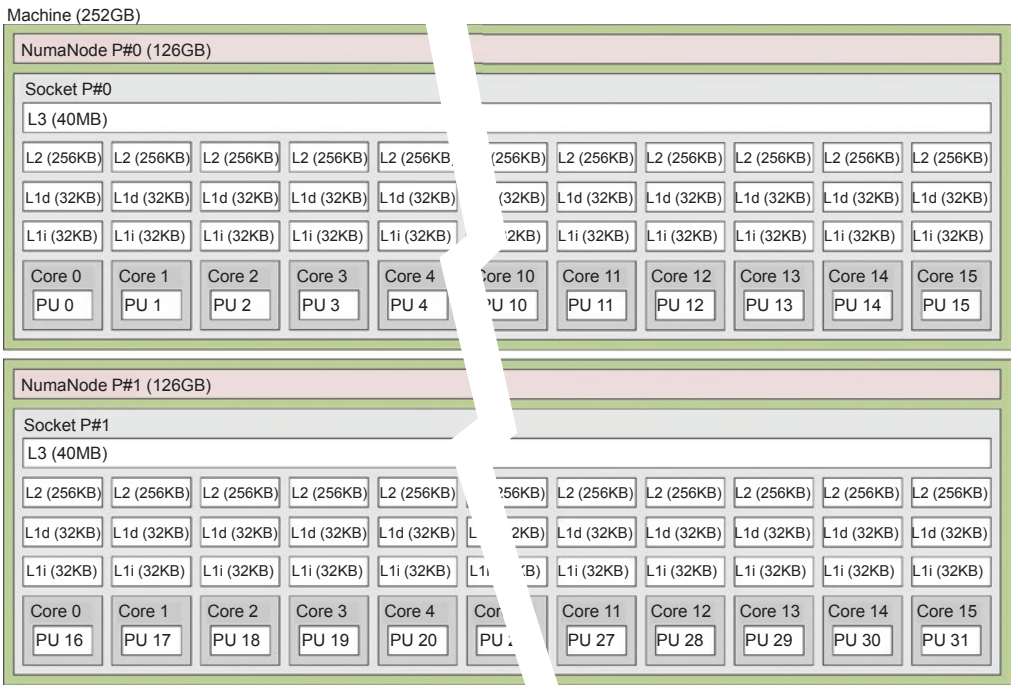

Figure 20-4. *Result of executing* `lstopo` *on aloe*

We see that on aloe each physical core includes a single logical core, numbered from 0-15 in the first domain and from 16-31 in the second one.

Understanding the Costs of Accessing Memory

Now that we know the topology of our platform, let's quantify the overhead due to nonlocal accesses assuming we already control processor affinity and data placement. Actually, we do control these two aspects on already available benchmarks, like `likwid-bench` available in the `likwid` tool. Using this benchmark, we can run the STREAM triad code (see the previous two chapters) using a single command line:

```
likwid-bench -t stream -i 1 -w S0:12GB:16-0:S0,1:S0,2:S0
```

which runs a single iteration (`-i 1`) of the stream benchmark configured with `-w` argument so that

- S0: The threads are pinned to the NUMA node 0.
- 12 GB: The three triad arrays occupy 12 GB (4 GB per array).
- 16: 16 threads will share the computation, each one processing chunks of 31,250,000 doubles (this is, 4000 million bytes/8 bytes per double/16 threads).
- `0:S0,1:S0,2:S0`: The three arrays are allocated on the NUMA node 0.

On yuca, the result of this command reports a bandwidth of 8219 MB/s. But it is a no-brainer to change the data placement for the three arrays, for example to the NUMA node 1 (using `0:S1,1:S1,2:S1`) keeping the computation by 16 threads confined in the NUMA node 0. Not surprisingly, the bandwidth we get now is only 5110 MB/s, which means we are losing a 38% of the bandwidth we measured when exploiting NUMA locality. We get similar results for other configurations that compute local data (data placement on the cores where the threads are pinned) and configurations that do not exploit locality (data placement on cores that do not have the thread affinity). On yuca, all nonlocal configurations result in the same bandwidth hit, but there are other NUMA topologies on which we pay different penalties depending on where the data is placed and where the threads are running.

On aloe we only have two NUMA nodes 0 and 1. Having the data and the computation on the same domain gives us 38671 MB/s, whereas going down the wrong path results in only 20489 MB/s (almost half, exactly 47% less bandwidth). We are certain that a reader like you, eager to read and learn about performance programming topics, is now motivated to exploit NUMA locality in your own projects!

Our Baseline Example

Figure 20-5 shows a parallel version of the triad example that we have been using recently, with just a `parallel_for` algorithm.

```
int main(int argc, const char* argv[]) {

  int nth = (argc>1) ? atoi(argv[1]) : 4;
  size_t vsize = (argc>2) ? atoi(argv[2]) : 100000000;
  float alpha = 0.5;
  tbb::task_scheduler_init init{nth};

  std::unique_ptr<double[]> A{new double[vsize]};
  std::unique_ptr<double[]> B{new double[vsize]};
  std::unique_ptr<double[]> C{new double[vsize]};

  for(size_t i = 0; i < vsize; i++){
    A[i] = B[i] = i;
  }

  auto t=tbb::tick_count::now();
  tbb::parallel_for(tbb::blocked_range<size_t>{0, vsize},
    [&](const tbb::blocked_range<size_t>& r){
      for (size_t i = r.begin(); i < r.end(); ++i)
        C[i] = A[i] + alpha * B[i];
    });
  double ts = (tbb::tick_count::now() - t).seconds();

  std::cout << "Time: " << ts << " seconds; ";
  std::cout << "Bandwidth: "  << vsize*24/ts/1000000.0 << "MB/s\n";
  return 0;
}
```

Figure 20-5. *The baseline algorithm to evaluate and improve*

The last two lines of this code, which is not yet optimized for NUMA, reports the execution time and the obtained bandwidth. For the latter, the total number of bytes accessed is computed as `vsize` × 8 bytes/double × 3 access per array element (two loads and one store), and this is divided by the execution time and by one million (to convert to Mbytes per second). On yuca, this results in the following output when running with 32 threads and arrays of one giga-element:

```
./fig_20_05 32 1000000000
Time: 2.23835 seconds; Bandwidth: 10722.2MB/s
```

and on aloe:

```
./fig_20_05 32 1000000000
Time: 0.621695 seconds; Bandwidth: 38604.2MB/s
```

Note that the bandwidth obtained with our triad implementation should not be compared with the one reported previously by `likwid-bench`. Now we are using 32 threads (instead of 16) that, depending on the OS scheduler, can freely run on every core (instead of confined to a single NUMA node). Similarly, arrays are now placed by the OS following its own data placement policy. In Linux, the default policy[2] is "local allocation" in which the thread that does the allocation determines the data placement: in local memory if there is enough space, or remote otherwise. This policy is sometimes called "first touch," because data placement is not done at allocation time, but at first touch time. This means that a thread can allocate a region but the thread that first accesses this region is the one raising the page fault and actually allocating the page on memory local to that thread. In our example of Figure 20-5, the same thread allocates and initializes the arrays, which means that the `parallel_for` worker threads running on the same NUMA node will have faster access. A final difference is that `likwid-bench` implements the triad computation in assembly language which prevents further compiler optimizations.

Mastering Data Placement and Processor Affinity

Binding data and computation is not trivial at all. Mainly because it depends on the Operating System and each one has its own system calls. In Linux, the low-level interface is provided by `libnuma`[3] which includes functions to control the data placement and processor affinity policies implemented in the Linux kernel. A higher-level alternative is the `numactl`[4] command that tackles the same problem, offering less flexibility though.

However, it is not the best idea in the world to ruin the portability of our TBB application marrying to an OS dependent NUMA library. A portable and widely used alternative is the already mentioned `hwloc` library. Currently, TBB does not offer its own API to deal with NUMA locality, but as we will see later, there are measures we can take to get our TBB tasks to access local data when possible. At the time of writing this,

[2]We can query the enforced NUMA policy using numactl --show.
[3]http://man7.org/linux/man-pages/man3/numa.3.html.
[4]http://man7.org/linux/man-pages/man8/numactl.8.html.

manual control of data placement and processor affinity has to be done via a third-party library, and without loss of generality, we will resort to `hwloc` in this chapter. This library can be used in Windows, MacOS, and Linux (actually, in Linux `hwloc` uses `numactl/libnuma` underneath).

In Figure 20-6, we present an example that queries the number of NUMA nodes and then allocates some data on each node to later create a thread per node and bind it to the corresponding domain. We are using `hwloc` 2.0.1 in the following.

```
#include <hwloc.h>
…
int main(void)
{
  hwloc_topology_t topo;
  hwloc_topology_init(&topo);
  hwloc_topology_load(topo);

  //* Print the number of NUMA nodes.
  int num_nodes = hwloc_get_nbobjs_by_type(topo, HWLOC_OBJ_NUMANODE);
  std::cout << "There are " << num_nodes << " NUMA node(s)\n";

  double ** data = new double*[num_nodes];
  //* Allocate some memory on each NUMA node
  long size = 1024*1024;
  alloc_mem_per_node(topo, data, size);

  sout_t sout;
  //* One master thread per NUMA node
  alloc_thr_per_node(topo, sout);

  for (auto &s : sout) {
    std::cout << s.str();
  }
  //* Free the allocated data and topology
  for(int i = 0; i < num_nodes; i++){
    hwloc_free(topo, data[i], size);
  }
  hwloc_topology_destroy(topo);
  delete [] data;
  return 0;
}
```

Figure 20-6. *Using* `hwloc` *to allocate memory and bind threads to each NUMA node*

A recurrent argument of all hwloc functions is the object topology, topo in our example. This object is first initialized and then loaded with the available information of the platform. After that, we are ready to get information from the topo data structure, as we do with hwloc_get_nbobjs_by_type that returns the number of NUMA nodes when the second argument is HWLOC_OBJ_NUMANODE (several other types are available, as HWLOC_OBJ_CORE or HWLOC_OBJ_PU – logical core or processing unit). This number of NUMA nodes is stored in the variable num_nodes.

The example continues by creating an array of num_nodes pointers to doubles that will be initialized inside the function alloc_mem_per_node. The function call to alloc_thr_per_node creates num_nodes threads, each one pinned to the corresponding NUMA node. These two functions are described in Figures 20-7 and 20-8, respectively. The example finishes by freeing the allocated memory and the topo data structure.

```
void alloc_mem_per_node(hwloc_topology_t topo,
                        double **data,
                        long size){
  int num_nodes = hwloc_get_nbobjs_by_type(topo, HWLOC_OBJ_NUMANODE);
  for(int i = 0; i < num_nodes; i++){ //for each NUMA node
    hwloc_obj_t numa_node = hwloc_get_obj_by_type(topo,
                                                  HWLOC_OBJ_NUMANODE, i);
    char *s;
    hwloc_bitmap_asprintf(&s, numa_node->cpuset);

    std::cout<<"NUMA node " << i << " has cpu bitmask: " << s <<'\n';
    free(s);
    hwloc_bitmap_asprintf(&s, numa_node->nodeset);
    std::cout << "Allocate data on node " << i
              << " with node bitmask " << s << '\n';
    free(s);

    data[i] = (double *) hwloc_alloc_membind(topo,
      size*sizeof(double), numa_node->nodeset,
      HWLOC_MEMBIND_BIND, HWLOC_MEMBIND_BYNODESET);
  }
}
```

Figure 20-7. *Function that allocates an array of doubles per NUMA node*

Figure 20-7 shows the implementation of the function alloc_mem_per_node. The key operations are hwloc_get_obj_by_type that returns a handle to the ith NUMA node object, numa_node, when the second and third arguments are HWLOC_OBJ_NUMANODE

and `i`, respectively. This `numa_node` has several attributes like `numa_node->cpuset` (a bitmask identifying the logical cores included in the node) and `numa_node->nodeset` (a similar bitmask that identifies the node). The function `hwloc_bitmap_asprintf` comes in handy to translate these sets into strings as we will see latter in the output of the program. Using the `nodeset` bitmask, we can allocate memory in a node with `hwloc_alloc_membind`.

The output we get on yuca when running the code until `alloc_mem_per_node` returns to the main function is

```
There are 4 NUMA node(s)
NUMA node 0 has cpu bitmask: 0x000000ff,0x000000ff
Allocate data on node 0 with node bitmask 0x00000001
NUMA node 1 has cpu bitmask: 0x0000ff00,0x0000ff00
Allocate data on node 1 with node bitmask 0x00000002
NUMA node 2 has cpu bitmask: 0x00ff0000,0x00ff0000
Allocate data on node 2 with node bitmask 0x00000004
NUMA node 3 has cpu bitmask: 0xff000000,0xff000000
Allocate data on node 3 with node bitmask 0x00000008
```

where we see the `cpuset` and `nodeset` of each NUMA node. If we refresh our memory looking again at Figure 20-3, we see that in node 0 we have eight cores with 16 logical cores, numbered from 0 to 7 and from 32 to 39, which are represented in `hwloc` with the bitmask `0x000000ff,0x000000ff`. Note that the "," separates the two sets of logical cores sharing the eight physical ones. To compare with a Hyperthreading disabled platform, this is the corresponding output on aloe:

```
There are 2 NUMA node(s)
NUMA node 0 has cpu bitmask: 0x0000ffff
Allocate data on node 0 with node bitmask 0x00000001
NUMA node 1 has cpu bitmask: 0xffff0000
Allocate data on node 1 with node bitmask 0x00000002
```

In Figure 20-8, we list the function `alloc_thr_per_node` that spawns a thread per NUMA node and then bind it using the `cpuset` attribute.

```
using sout_t = tbb::enumerable_thread_specific<std::stringstream>;
void alloc_thr_per_node(hwloc_topology_t topo, sout_t& sout){
  int num_nodes = hwloc_get_nbobjs_by_type(topo, HWLOC_OBJ_NUMANODE);
  std::vector<std::thread> vth;
  for(int i = 0; i < num_nodes; i++){
    vth.push_back(std::thread{[i, num_nodes, &topo, &sout]()
    {
      int err;
      sout.local() << "I'm masterThread: " << i << " out of "
                   << num_nodes << '\n';
      sout.local() << "Before: Thread: " << i
                   << " with tid " << std::this_thread::get_id()
                   << " on core " << sched_getcpu() << '\n';

      hwloc_obj_t numa_node = hwloc_get_obj_by_type(topo,
                                          HWLOC_OBJ_NUMANODE,i);
      err = hwloc_set_cpubind(topo, numa_node->cpuset,
                              HWLOC_CPUBIND_THREAD);
      assert(!err);
      sout.local() << "After: Thread: " << i
                   << " with tid " << std::this_thread::get_id()
                   << " on core " << sched_getcpu() << '\n';
    }});
  }
  for(auto &th: vth) th.join();
}
```

Figure 20-8. *Function that creates and pins a thread per NUMA node*

This function also queries the number of NUMA nodes, `num_nodes`, to later iterates this number of times inside a loop that creates the threads. In the lambda expression that each thread executes, we use `hwloc_set_cpubind` to bind the thread to each particular NUMA node, now relying on the `numa_node->cpuset`. To validate the pinning, we print the thread id (using `std::this_thread::get_id`) and the id of the logical core on which the thread is running (using `sched_getcpu`). The result on yuca is next, also illustrated in Figure 20-9.

```
Before: Thread 0 with tid 873342720 on core 33
After: Thread 0 with tid 873342720 on core 33
Before: Thread 1 with tid 864950016 on core 2
After: Thread 1 with tid 864950016 on core 8
Before: Thread 2 with tid 856557312 on core 33
After: Thread 2 with tid 856557312 on core 16
Before: Thread 3 with tid 848164608 on core 5
After: Thread 3 with tid 848164608 on core 24
```

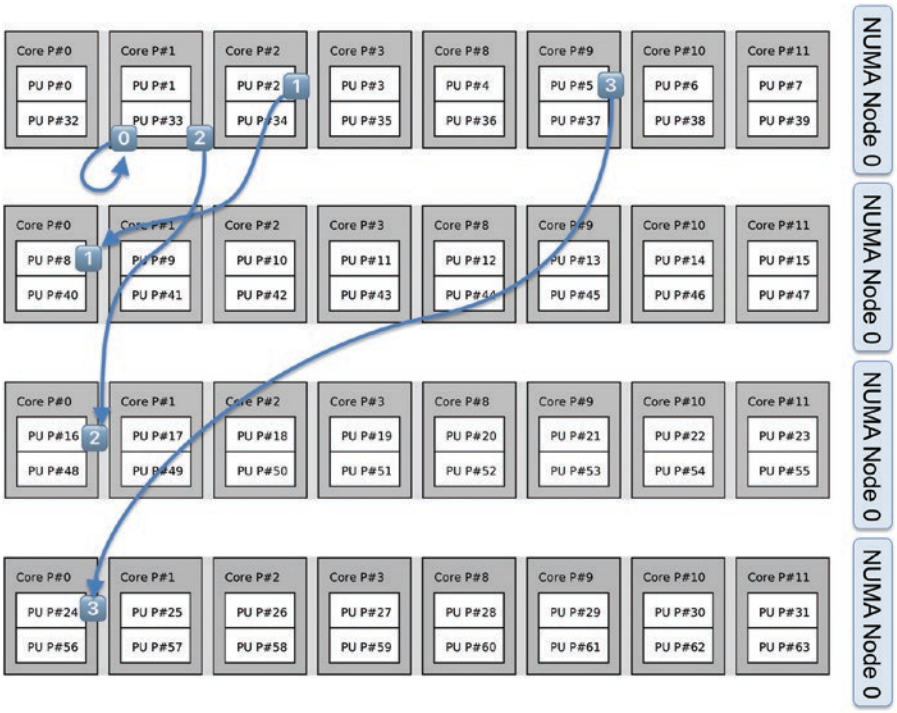

Figure 20-9. *Depicting the movement of threads due to pinning to NUMA nodes on yuca*

Two things are worth mentioning here. First, the threads are initially allocated by the OS on logical cores in the same NUMA node, since it assumes they will collaborate. Threads 0 and 2 are even allocated on the same logical core. Second, the threads are not pinned to a single core, but to the whole set of cores belonging to the same NUMA node. This allows for some leeway if the OS considers it better to move a thread to a different core of the same node. For completeness, here is the equivalent output on aloe:

```
Before: Thread: 0 with tid 140117643171584 on core 3
After: Thread: 0 with tid 140117643171584 on core 3
Before: Thread: 1 with tid 140117634778880 on core 3
After: Thread: 1 with tid 140117634778880 on core 16
```

There are many more features of `hwloc` and `likwid` that the interested reader can learn from the respective documentation and online tutorials. However, what we have covered in this section suffices to move on, roll up our sleeves, and implement a NUMA-conscious version of the triad algorithm using TBB.

Putting hwloc and TBB to Work Together

Clearly, the overarching goal is to minimize the number of nonlocal accesses, which implies conducting the computation on the cores nearest to the memory in which the data is stored. A quite simple approach is to manually partition the data on the NUMA nodes and confine the threads that process this data to the same nodes. For educational purposes, we will first describe this solution and in the next section briefly elaborate on more advanced alternatives.

We can rely on the hwloc API to accomplish the data placement and processor affinity tasks, but we want a NUMA-aware TBB implementation of the triad benchmark. In this case, the TBB scheduler is the one managing the threads. From Chapter 11, we know that a number of threads are created inside the `tbb::task_scheduler_init` function. Additionally, this TBB function creates a default arena with enough worker slots to allow the threads to participate in executing tasks. In our baseline implementation of triad (see Figure 20-5), a `parallel_for` takes care of partitioning the iteration space into different tasks. All the threads will collaborate on processing these tasks, irrespectively of the chunk of iterations that each task processes and of the core on which the thread is running. But we don't want that on a NUMA platform, right?

Our simplest alternative to the baseline triad implementation will enhance the implementation by performing the following three steps:

- It will partition and allocate the three vectors, A, B, and C, of the triad algorithm on the different NUMA nodes. As the simplest solution, a static block partitioning will do for now. On yuca, this means that four big chunks of A, B, and C will be allocated on each one of the four nodes.
- It will create a master thread on each NUMA node. Each master thread will create its own task arena and its own local `task_scheduler_observer`. Then, each master thread executes its own `tbb::parallel_for` algorithm to process the fraction of A, B, and C that correspond to this NUMA node.
- It will automatically pin the threads that join each arena to the corresponding NUMA node. The local `task_scheduler_observer` that we create for each arena will take care of this.

Let's see the implementation of each one of the described bullet points. For the main function we slightly modify the one we presented for the hwloc example of Figure 20-6. In Figure 20-10, we list the new lines required for this new example, using ellipsis (…) on the lines that do not change.

```
int main(int argc, char** argv)
{
  int thds_per_node = (argc>1) ? atoi(argv[1]) : 4;
  size_t vsize = (argc>2) ? atoi(argv[2]) : 100000000;
  …
  double **data = new double*[num_nodes];
  times = std::vector<double>(num_nodes);
  //* Allocate some memory on each NUMA node
  long doubles_per_node = vsize*3/num_nodes;
  alloc_mem_per_node(topo, data, doubles_per_node);

  //* One master thread per NUMA node
  tbb::task_scheduler_init init{(thds_per_node-1)*num_nodes};
  auto t = tbb::tick_count::now();
  alloc_thr_per_node(topo, data, vsize/num_nodes, thds_per_node);
  double ts = (tbb::tick_count::now() - t).seconds();
  …
  delete [] data;
  return 0;
}
```

Figure 20-10. *Main function of the NUMA-conscious implementation of triad*

The program argument, thds_per_node, allows us to play with different number of threads per NUMA node. As in the example of Figure 20-6, num_nodes is the number of NUMA nodes that we obtain using the hwloc API. Consequently, we pass to the TBB scheduler constructor (thds_per_node-1)*(num_nodes) instead of thds_per_node*num_nodes because we will explicitly create the additional num_nodes master threads inside alloc_thr_per_node.

The function alloc_mem_per_node is essentially the same as the one listed in Figure 20-7, but now it is called with a different size argument: doubles_per_node = vsize*3/num_nodes, where vsize is the size of the three vectors, so the total amount of doubles is multiplied by 3, but divided by the number of nodes to implement the block partitioning. For the sake of cleanness, we assume that vsize is a multiple of num_nodes. At the completion of alloc_mem_per_node, data[i] points to the data allocated on the ith NUMA node.

There are other differences in the adapted version of the `alloc_thr_per_node` function as we see in Figure 20-11. It now receives a handle to the data, the size of the local vectors that will be traversed per node, `lsize`, and the number of threads per node set by the user, `thds_per_node`.

```
void alloc_thr_per_node(hwloc_topology_t topo, double** data,
                        size_t lsize, int thds_per_node){
  float alpha = 0.5;
  int num_nodes = hwloc_get_nbobjs_by_type(topo, HWLOC_OBJ_NUMANODE);
  std::vector<std::thread> vth;
  for(int i = 0; i < num_nodes; i++){
    vth.push_back(std::thread{
      [=,&topo](){
        hwloc_obj_t numa_node = hwloc_get_obj_by_type(topo,
                                              HWLOC_OBJ_NUMANODE,i);
        int err = hwloc_set_cpubind(topo, numa_node->cpuset,
                                    HWLOC_CPUBIND_THREAD);
        assert(!err);
        double *A = data[i];
        double *B = data[i] + lsize;
        double *C = data[i] + 2*lsize;

        for(size_t j = 0; j < lsize; j++){
          A[j] = B[j] = j;
        }
        tbb::task_arena numa_arena{thds_per_node};
        PinningObserver p{numa_arena, topo, i, thds_per_node};
        auto t = tbb::tick_count::now();
        numa_arena.execute([&](){
          tbb::parallel_for(tbb::blocked_range<size_t>{0, lsize},
            [&](const tbb::blocked_range<size_t>& r){
              for (size_t i = r.begin(); i < r.end(); ++i)
                C[i] = A[i] + alpha * B[i];
            });
        });
        double ts = (tbb::tick_count::now() - t).seconds();
        times[i] = ts;
      }
    });
  }
  for (auto &th: vth) th.join();
}
```

Figure 20-11. *Function that creates a thread per node to compute the triad computation on each NUMA node*

Note that in the code snippet presented in Figure 20-11, inside the `i`-loop that traverses the `num_nodes`, there are three nested lambda expressions: (1) for the thread object; (2) for `task_arena::execute` member function; and (3) for the `parallel_for` algorithm. In the outer one, we first pin the thread to the corresponding NUMA node, `i`.

The second step is to initialize the pointers to arrays `A`, `B`, and `C` that were allocated in the `data[i]` array. In Figure 20-10, we call `alloc_thr_per_node` using as the third argument `vsize/num_nodes` because on each node we traverse just one chunk of the block distribution of the three arrays. Hence, the function's argument `lsize = vsize/num_nodes`, which is used in the loop that initializes arrays `A` and `B` and as the argument to the `parallel_for` that computes `C`.

Next, we initialize a per NUMA node arena, `numa_arena`, that is later passed as argument to a `task_scheduler_observer` object, `p`, and used to invoke a `parallel_for` confined to this arena (using `numa_arena.execute`). Here lies the key of our NUMA-aware implementation of triad.

The `parallel_for` will create tasks that traverse chunks of the local partition of the three vectors. These tasks will be executed by threads running on the cores of the same NUMA node. But up to now, we just have `thds_per_node*num_nodes` threads, out of which `num_nodes` have been explicitly spawned as master threads and pinned to a different NUMA node, but the rest are still free to run everywhere. The threads that are available in the global thread pool will each join one of the `num_nodes` arenas. Conveniently, each `numa_arena` has been initialized with `thds_per_node` slots, one already occupied by a master thread and the rest available for worker threads. Our goal now is to pin the first `thds_per_node-1` threads that enter each `numa_arena` to the corresponding NUMA node. To that end, we create a `PinningObserver` class (deriving from `task_scheduler_observer`) and construct an object, `p`, passing four arguments to the constructor: `PinningObserver p{numa_arena, topo, i, thds_per_node}`. Remember that here, `i` is the id of the NUMA node for the master thread `i`.

In Figure 20-12, we see the implementation of the `PinningObserver` class.

```
class PinningObserver : public tbb::task_scheduler_observer {
  hwloc_topology_t topo;
  hwloc_obj_t numa_node;
  int numa_id;
  int num_nodes;
  tbb::atomic<int> thds_per_node;
  tbb::atomic<int> masters_that_entered;
  tbb::atomic<int> workers_that_entered;
  tbb::atomic<int> threads_pinned;
public:
  PinningObserver(tbb::task_arena& arena, hwloc_topology_t& _topo,
                  int _numa_id, int _thds_per_node)
                  : task_scheduler_observer{arena}, topo{_topo},
                    numa_id{_numa_id}, thds_per_node{_thds_per_node}{
    num_nodes = hwloc_get_nbobjs_by_type(topo,
                                         HWLOC_OBJ_NUMANODE);
    numa_node = hwloc_get_obj_by_type(topo,
                                      HWLOC_OBJ_NUMANODE,numa_id);
    masters_that_entered = 0;
    workers_that_entered = 0;
    threads_pinned = 0;
    observe(true);
  }
  void on_scheduler_entry(bool is_worker) {
    if (is_worker) ++workers_that_entered;
    else ++masters_that_entered;
    if(--thds_per_node > 0){
      int err = hwloc_set_cpubind(topo, numa_node->cpuset,
                                  HWLOC_CPUBIND_THREAD);
      assert(!err);
      threads_pinned++;
    }
  }
};
```

Figure 20-12. *Implementation of the local* `task_scheduler_observer` *for triad*

The `task_scheduler_observer` class was introduced in Chapter 13. It has a preview feature that allows us to have an observer per task arena – also called a local `task_scheduler_observer`. This kind of observer is initialized with a reference to the arena, as we do in the initializer list of the `PinningObserver` constructor using `task_scheduler_observer{arena}`. This results in the execution of the member function `on_scheduler_entry` for each thread that enters this particular arena. The constructor of the class also sets the number of NUMA nodes, `num_nodes`, and the `numa_node` object that will give us access to the `numa_node->cpuset` bitmask. The constructor finally calls the member function `observe(true)` to start tracking whether or not a task enters the arena.

The function `on_scheduler_entry` keeps track of the number of threads that have been already pinned to the `numa_node` in the atomic variable `thds_per_node`. This variable is set in the initializer list of the constructor to the number of threads per node that the user pass as the first argument of the program. This variable is decremented for each thread entering the arena, which will get pinned to the node only if the value is greater than 0. Since each `numa_arena` was initialized with `thds_per_node` slots, and the already pinned master thread that creates the arena occupies one of the slots, the `thds_per_node - 1` threads that join the arena first will get pinned to the node and work on tasks generated by the `parallel_for` that this arena is executing.

Note The implementation of our PinningObserver class is not totally correct. One thread may leave the arena and reenter the same arena, getting pinned twice, but decrementing the number `thds_per_node`. A more correct implementation would check that the thread entering the arena is a new one that has not been pinned to this arena already. To avoid complicating the example, we leave this correction as an exercise to the reader.

We can now assess on yuca and aloe the bandwidth (in Mbytes per second) of this NUMA optimized version of the triad algorithm. To compare with the baseline implementation in Figure 20-5, we set the vector sizes to 10^9 doubles and set the number of threads per NUMA node so that we end up with 32 threads total. For example, in yuca we call the executables as follows:

```
baseline:          ./fig_20_05 32 1000000000
NUMA conscious:    ./fig_20_10  8 1000000000
```

The results presented in the table of Figure 20-13 are the average of ten runs in which yuca and aloe had a single user that was using the platform exclusively to conduct the experiments.

	baseline	NUMA conscious	
	Bandwidth (MBytes/s)	Bandwidth (MBytes/s)	Speedup
yuca	10722	18652	1.74
aloe	38604	59659	1.54

Figure 20-13. *Speedup due to the NUMA-conscious implementation*

This is 74% faster execution on yuca, and 54% on aloe! Would you ignore this extra amount of performance that we can squeeze out of a NUMA architecture with some extra implementation work?

To further investigate this improvement, we can take advantage of the `likwid-perfctr` application that is able to read out hardware performance counters. Invoking `likwid-perctr -a`, we get a list of groups of events that can be specified using only the group name. In aloe, `likwid` offers a `NUMA` group, which collects information about local and remote memory accesses. To measure the events in this group on our baseline and NUMA-conscious implementations, we can invoke these two commands:

```
likwid-perfctr -g NUMA ./fig_20_05 32 1000000000
likwid-perfctr -g NUMA ./fig_20_10 16 1000000000
```

This will report plenty of information about the value of some performance counters on all the cores. Among the counted events are

```
OFFCORE_RESPONSE_0_LOCAL_DRAM
OFFCORE_RESPONSE_1_REMOTE_DRAM
```

which give us approximate information (because is based on event-based sampling) of the amount of data accessed in local memory and remote memory. For the baseline triad implementation, the ratio of local data over remote data is only 3.25, but it raises up to 25.5 in the NUMA optimized triad-numa version. This confirms that, for this memory bound application, the effort we made to exploit NUMA locality pays off in terms of both the amount of local accesses and consequently the execution bandwidth.

More Advanced Alternatives

For the regular triad code, the simple solution we have implemented is okay, but TBB's work-stealing scheduler is confined to balancing the load on each NUMA node independently. On yuca, there will be four `parallel_for` algorithms running, each on a NUMA node with eight threads served by eight physical cores. The downside of our simple approach is that the four arenas have been configured with eight slots, which is okay for the steady-state part of the execution, but limits TBB's flexibility if the load is not perfectly balanced between the NUMA nodes.

For example, if one of the `parallel_for` algorithms ends first, eight threads become idle. They come back to the global thread pool but cannot join any of the other three busy arenas because all the slots are already filled. A simple solution for this involves increasing the number of slots of the arenas, while keeping the number of pinned threads to `thds_per_node`. In such a case, if a `parallel_for` finishes first, the eight threads returning to the global pool can be redistributed in the free slots of the other three arenas. Note that these threads are still pinned to the original node, although they will work now in a different arena of a different node and therefore memory accesses will be remote.

We could pin the threads entering the extended arena to the corresponding NUMA node when they occupy its free slots (even if they were pinned to a different NUMA node before). Now these helping threads will also access local memory. However, the node can become oversubscribed, which usually hurts performance (if not, you should oversubscribe every NUMA node from the very beginning). For each particular application and architecture, thorough experimentation should be carried out to decide whether it is advantageous to migrate the thread to the NUMA node or to remotely access the data from the original node. For the simple and regular triad algorithm, none of these discussed approaches significantly improves the performance, but in more complex and irregular applications they might. Not only do remote access have overhead, but also thread migration from one arena to another, as well as pinning the thread once again, represent an overhead that has to be amortized by better load balancing of the work.

Another battle that we can choose to fight concerns the data partitioning. We used a basic block distribution of the three arrays in our simple triad implementation, but we certainly know of better data distributions for more irregular applications. For example, instead of partitioning upfront the iteration space among the NUMA nodes, we can follow a guided scheduling approach. Each master thread leading the computation on each NUMA node can get larger chunks of the iteration space at the beginning of the computation and smaller as we approach the end of the space. The caveat here is to guarantee that chunks have enough granularity to be repartitioned again among the cores of each NUMA node.

A more elaborate alternative consists in generalizing the work-stealing framework in a hierarchical way. In order to allow work stealing both between arenas and within each arena, a hierarchy of arenas can be implemented. A similar idea was implemented for Cilk by Chen and Guo (see the “For More Information” section) who proposed a triple-level work-stealing scheduler that yielded up to 54% of performance improvement over

more traditional work-stealing alternatives for memory-bound applications. Note that memory-bound applications will benefit more from NUMA locality exploitation than CPU-bound ones. For the latter, memory access overhead is usually hidden by CPU intensive computations. Actually, for CPU-bound applications, adding extra complexity to the scheduler in order to exploit NUMA locality can result in an extra overhead that ends up not paying off.

Summary

In this chapter, we explored some alternatives to exploit NUMA locality combining TBB and third-party libraries that help in controlling the data placement and processor affinity. We began by studying the enemy that we want to defeat: the NUMA architecture. To that end, we introduced some ally libraries, `hwloc` and `likwid`. With them we can not only query the low-level details of the NUMA topology but also control data placement and processor affinity. We illustrated the use of some of the `hwloc` functions to allocate per-node memory, create one thread per NUMA node and pin threads to the cores of the node. With this template, we re-implemented a baseline version of the triad algorithm, now paying attention to NUMA locality. The simplest solution consisted of distributing the three triad arrays in blocks that are allocated and traversed in the different NUMA nodes. The library `hwloc` was key to allocate and pin the threads, and the TBB `task_arena` and `task_scheduler_observer` classes were instrumental in identifying the threads entering a particular NUMA node. This initial solution is good enough for a code as regular as the triad benchmark, which reports 74% and 54% performance improvement (with respect to the baseline triad implementation) on two different NUMA platforms, respectively. For more irregular and complex applications, more advanced alternatives are sketched in the last section of the chapter.

For More Information

Here are some additional reading materials we recommend related to this chapter:

- Christoph Lameter, NUMA (Non-Uniform Memory Access): An Overview, ACMqueue, Volume 11, issue 7, 2013.
- Ulrich Drepper, What Every Programmer Should Know About Memory, `www.akkadia.org/drepper/cpumemory.pdf`, 2017.

- Quan Chen, Minyi Guo and Haibing Guan, LAWS: Locality-Aware Work-Stealing for Multi-socket Multi-core Architectures, International Conference on Supercomputing, ICS, 2014.