4 Thread Affinity

The topic of this chapter is how to leverage the hardware characteristics of the system by controlling the placement of OpenMP threads. This allows the user to optimize for memory bandwidth, memory latency, or cache utilization and is referred to as optimizing for *thread affinity*, or just *affinity*, for short.

OpenMP 4.0 introduced a portable, small, powerful set of features and constructs to support this [8]. In OpenMP 4.5, this was extended with additional runtime support. In this chapter a complete overview of the support for thread affinity is given.

4.1 The Characteristics of a cc-NUMA Architecture

Until relatively recently, only very large shared memory systems had a cc-NUMA architecture, but nowadays, even many small two socket, general purpose systems have this kind of memory system.

The reason this architecture became so dominant is because the number of cores in a processor, as well as the number of sockets, continues to increase. A monolithic memory interconnect, with a fixed memory bandwidth, quickly becomes the bottleneck.

This may be avoided by physically distributing the memory. In a cc-NUMA design, each socket connects to a subset of the total memory in the system. A cache coherent interconnect glues all the sockets together and provides a single system image to the user. Such a memory subsystem is much more scalable, because the aggregate memory bandwidth scales with the number of sockets in the system.

The benefit of the special interconnect is that an application has transparent access to all of the memory in the system, regardless of where the data resides. There is a price to pay though. The time to access data (and instructions for that matter) is no longer uniform: it depends where that data is located in the system.

In Figure 4.1, a generic cc-NUMA architecture is shown. This is a template system with all the processor and memory components found in a contemporary, general purpose, multi-socket system. Throughout the remainder of this chapter it is used to illustrate various features and examples.

This system has two sockets. Each socket has two cores with four hardware threads per core. Each core has its own set of L1 caches. The L1 caches are connected to a shared L2 cache, which is connected to the memory on the socket. The memory access time within a socket is uniform.

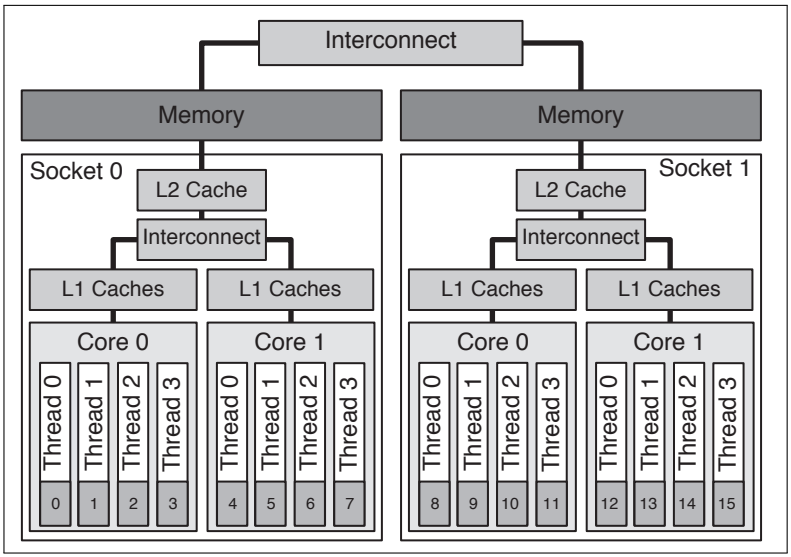

Figure 4.1: **A typical multi-core based server architecture** – This system consists of two sockets. There are two cores per socket and each core has four hardware threads. The memory is distributed across the system, but all of the memory may be accessed from anywhere in the system. The access time is non-uniform however. In the diagram, each hardware thread has a reference number, ranging from 0 to 15.

The two sockets are connected through a cache coherent interconnect. Memory access is transparent across the entire system, but it takes longer to access data on a remote socket.

The hardware threads are the execution vehicles. These are the units that execute an OpenMP thread. For reference purposes, they are numbered consecutively from 0 to 15. This is the number shown in the diagram for each hardware thread.

Unless otherwise noted, in the remainder of this chapter, reference to a computer system is implied to be a cc-NUMA system. On a system with a uniform, or "flat," memory access time, most of the topics covered in this chapter need not be considered. Such systems are however very rare these days.

4.2 First Touch Data Placement

Before thread affinity in OpenMP is discussed, it is important to know how the Operating System (OS) decides where to place the data.

Regardless where a thread executes, in the good case, data comes from the memory directly connected to the socket where the thread executes. In the worse case, data has to come from a memory connected to a socket far away in the system. This results in a longer, and possibly, non-uniform memory access time.

A non-uniform memory access time adds another dimension to application tuning. The goal is to service most, if not all, data accesses from the local memory connected to the socket where the thread is running. This maximizes bandwidth, and reduces the latency by avoiding relatively expensive accesses to a remote memory.

The OS decides where to run the application threads, but it has no knowledge of the application characteristics. This makes it hard to decide what the best placement is and a default thread placement strategy is used, which may, or may not, be optimal.

The OS manages data at the page level. A page consists of a relatively large chunk of data, for example 4 or 8 Kbyte.[1] The decision where to place data, or better said, the pages containing the data, is made by the OS. The question is when and how it decides to do so. This is controlled by the page placement policy.

4.2.1 The Pros and Cons of First Touch Data Placement

With the *First Touch* placement policy, the thread (or process) that "touches" the data for the first time, has ownership of the corresponding page. This defines the *home* node for that page. More specifically, the first thread that accesses a page, memory capacity issues aside, has the data allocated in the memory closest to this thread.

There is no right or wrong when it comes to the placement policy, but the First Touch policy is a very reasonable default. It ensures that serial applications get the data in their local memory, guaranteeing the minimal data access time for a serial application.

First Touch not only delivers the best performance from the memory system, but also preserves the performance, relative to a comparable system with a flat memory

[1] It is often possible to increase the size of a page. There may be valid performance reasons to do so, but this discussion is outside the scope of this book.

architecture. This is the reason general purpose Operating Systems use this policy as a default.

For exactly the same reason it works so well for serial programs, this policy can be a performance bottleneck for a shared memory parallel application. The only thing that matters for the page placement is which thread accessed the data for the first time. First Touch, for example, works well if each thread in a parallel region initializes a new block of memory it just allocated.

Things may not always be so easy though. If, for example, the master thread initializes shared data, all other threads need to access it from its memory. As long as this data is read-only, data caches may help to reduce the cost of a memory access, but in case the data is modified, the cache lines move around. This degrades performance.

I/O is another possible problem. If data is read into data structures that have not been used before, the pages are initialized upon reading the data. Because in most cases only one thread performs the read operation, the same placement problem occurs.

These types of placement issues create a load imbalance, because a certain percentage of the threads run slower due to a longer memory access time. On top of that, a single memory controller receives most, or even all, of the requests. Altogether it means the available total bandwidth in the system is not fully utilized.

Luckily, there is a simple technique to turn First Touch placement into an advantage. If applicable, it provides for an easy and elegant solution.

4.2.2 How to Exploit the First Touch Policy

Knowing how the First Touch policy works, the solution is rather straightforward. Depending on the data access pattern in the application, the data initialization phase may need to be adapted. It has to be parallelized in such a way that each thread touches those portions of data it needs most. This is somewhat of a reversed engineering approach and works best if data access patterns are static. That is, if a thread mostly accesses the same part of the data.

In case the data access pattern is irregular, First Touch does not work well, because there is no data locality. One approach to improve the situation, is to randomly place the pages using a system call, if supported by the OS.[2]

[2] The `madvise()` system call supported on most Operating Systems is an example.

```
 1 int *a;
 2     .....
 3 // Redundant parallel initialization to place the data upfront
 4 #pragma omp parallel for default(none) shared(n,a,b)
 5 for (int i=0; i<n; i++)
 6     a[i] = 0;
 7      .....
 8 // At this point, the placement has been decided upon already
 9 fread(a, sizeof(int), n, fp);
10      .....
11 // Reading the data takes advantage of the architecture
12 #pragma omp parallel for default(none) shared(n,a,b)
13 for (int i=0; i<n; i++)
14      b[i] = my_func(a[i]);
```

Figure 4.2: **An example to exploit First Touch** – The redundant parallel loop at lines $4-6$ ensures the pages are placed in the memories of the threads executing this loop. When reading from file at line 9, the elements of array `a` are placed in those memories. This ensures that the parallel bandwidth is maximized in the parallel loop at lines $12-14$.

This is also why the popular `calloc()` memory allocation function may not be the best choice. It is very convenient that this function allocates the memory, as well as initializes the data to zero. The potential drawback is that the latter may mean the pages end up in the wrong place.

A more cc-NUMA friendly solution is to use the regular `malloc()` function, followed by a parallelized data initialization phase.

In some cases, less intuitive approaches are required. For example, if it is not necessary to initialize the data, or in case it is not possible to parallelize the data initialization, the data placement may be non-optimal.

A solution is to insert an extra and *redundant* parallel data initialization, setting all data to zero, say. If this is performed prior to the first use of the data, the First Touch placement policy distributes the pages over the memories in the system.

An example is reading data from disk. This is often performed by a single thread, but this means that all this data ends up in the memory of the socket where the thread is executing. An extra parallel initialization is really beneficial here and illustrated in Figure 4.2.

Name	Set Definition
P_{00}	$\{0, 1, 2, 3\}$
P_{01}	$\{4, 5, 6, 7\}$
P_{10}	$\{8, 8, 10, 11\}$
P_{11}	$\{12, 13, 14, 15\}$

Figure 4.3: **Four set definitions based upon the hardware thread numbers** – Each set contains the hardware thread numbers of one of the cores from Figure 4.1.

In this example, the parallel redundant initialization at lines $4-6$ ensures the data is allocated in the memories of those threads executing the loop. In this way, the read operation at line 9 has no impact on the placement and the data is distributed automatically. The read operations in the parallel loop at lines $12-14$, utilizes the parallel memory architecture.

Another case where this kind of extra initialization works well, is if the data structure is initialized in a complex way, and/or the initialization is not performed in an isolated location in the application.

In summary, the First Touch placement policy controls where data is located. In the ideal scenario, the threads are close to the data they access most frequently.

4.3 The Need for Thread Affinity Support

A characteristic of a cc-NUMA system is the aggregated bandwidth. If an application is bandwidth hungry, it may be beneficial to spread the threads over the sockets and leverage the bandwidth each memory controller is able to deliver. In the opposite placement scenario, threads are kept close to each other to improve the cache utilization. In case threads exchange data frequently, sharing data via a cache is very efficient.

While the OS is unaware of such characteristics, the user usually knows more about the application. This knowledge may be leveraged to control where the OpenMP threads should run and improve the performance.

This can be achieved as follows. In the generic architecture in Figure 4.1, there are 16 hardware threads. Each one of these is a target location where an OpenMP thread may run. In Figure 4.3, these threads are organized into four disjoint sets. Within each set, the memory access time is uniform. This is also true for the two

sets with the same first index, for example P_{00} and P_{01}, but not between sets with a different first index.

With the set definitions, it is easy to define where the threads should run. Suppose we want to use eight threads. In case the threads must stay close to each other, either the combination of sets P_{00} and P_{01}, or sets P_{10} and P_{11}, achieve this. If total bandwidth is important, two sets with a different first index should be selected.

Thread affinity in OpenMP is built upon such sets. These sets, or *places* as they are called in OpenMP, are defined by the user. Through additional user-level controls, they are mapped onto the computational hardware resources in the system. The remainder of this chapter introduces and discusses this in great detail.

4.4 The OpenMP Thread Affinity Philosophy

On a cc-NUMA architecture, performance and scalability critically rely on where the threads get their data from. Ideally, this is from the memory closest to where the threads execute. It is tempting to expose the memory hierarchy to the application and provide controls to migrate the data, but there are significant drawbacks:

- The memory hierarchy is complex today, and even more so in the future.
- The characteristics are detailed and vary substantially across different computer systems.
- Moving data is very costly and the cost increases with the number of hops in the interconnect.

Instead of migrating data to the threads, the alternative is much more attractive. The data is located in a memory connected to a socket in the system. The goal is to provide a mechanism to get the threads to execute close to the data they access most frequently. Even if the threads are initially in the "wrong" place, they can be moved to the data. Although not without a performance penalty either, moving a thread is much cheaper than moving the data to the thread(s).

The thread affinity support in OpenMP is based upon sets, similar as introduced in the previous section. In OpenMP, such sets are called *places*. Through thread affinity it is also guaranteed that, for the duration of the parallel region, a thread stays in the place it has been assigned to, once it has migrated to its destination

place. Depending on the affinity policy selected on a subsequent parallel region, the threads may, or may not, migrate again.

This provides for a high level control mechanism to maximize thread affinity. All that needs to be done is to specify where in the system the OpenMP threads are allowed to execute and define their relative position. For example, to keep threads close to each other, or spread them over the system.

In OpenMP, the two key concepts that must be used to specify thread affinity, are the *places* and the *affinity policy*. In absence of either one of these, or both, implementation-defined default settings are used.

- **OpenMP Places** - A place is a hardware resource that can execute an OpenMP thread.

  There are different ways to specify places, but ultimately a list with such execution vehicles for an OpenMP thread to run on, is used by the OpenMP runtime scheduler to decide where to run the threads.

  Environment variable `OMP_PLACES` is used to describe the system in terms of such system resources.

  The easiest way to specify places is to use one of the symbolic names "sockets," "cores," or "threads." Although the interpretation of these words is implementation-defined, it is safe to assume they match the hardware characteristics suggested by the name.

  In addition to this, it is allowed for an implementation to support additional names to provide tailored support for architectural features not easily captured through these standard keywords.

  These symbolic names have several advantages and are recommended to use, but in some cases more explicit control over thread placement may be desired.

  This level of control is supported through an explicit *place list*. Such a list consists of one or more sets with integer numbers. The place list is defined at program start-up and used throughout the execution of the program. In other words, it is fixed.

These numbers are the scheduling units the OS uses to schedule OpenMP threads on. In the most common case, the numbers represent hardware threads, but they could also be related to GPU channels for example.[3]

The selection of the numbers, plus the definition of the places within a place list, provides sufficient flexibility to fully control thread placement.

Because there may be many numbers, resulting in a long list, the interval notation is very convenient. Through a compact notation, it simplifies the list, if a range of numbers is used. For example, the set defined by $\{0:4:8\}$ is equivalent to $\{0, 8, 16, 24\}$, but the notation is much shorter and less error-prone.

This, and much more about OpenMP places, is explained in detail in Section 4.5.1, starting on page 161.

- **Affinity Policy** - While the place list defines all the resources available to the OpenMP runtime scheduler, the affinity policy may be adjusted on each (nested) parallel region, and controls *how* the threads should be scheduled, relative to each other.

  Environment variable `OMP_PROC_BIND` is set to the appropriate policy, or policies, in case of nested parallel regions. If not set explicitly by the user, an implementation-defined setting is used.

  Within the application, the `proc_bind` clause may be used to specify the policy at any level in the (nested) parallel region. The policy set through the clause is not persistent. It is only valid for the parallel region the clause applies to. In the absence of this clause, the policies set through `OMP_PROC_BIND`, or the default settings if this variable is not defined, are applied. This is regardless of any explicit policy settings on previous parallel regions.

  The following policies are supported: `master`, `close` and `spread`.

  The names are descriptive of the behavior and can be used to schedule the threads on the same place where the master thread runs, keep them close to each other, or spread them out. This is all defined in terms of the places in the place list. The precise definitions of these three policies are covered in detail in Section 4.6, starting on page 168.

[3] In the remainder these numbers are assumed to represent hardware threads, but they can easily represent a different scheduling unit.

	Thread Affinity (OMP_PROC_BIND)		
	not set	**explicitly set**	
Definition of places (OMP_PLACES)		**false**	**true, master, close, or spread**
not set	system defaults	*affinity disabled*	default place list used
explicitly set	depends on default	*affinity disabled*	explicit settings used

Figure 4.4: **Interaction between OMP_PLACES and OMP_PROC_BIND** – This table summarizes the interaction between these two environment variables. As usual, the default values are system-dependent.

> Implied with an affinity policy is *thread binding*. Threads are not allowed to execute outside of the place they are assigned to.
>
> It is allowed to only request binding, without further specifying where threads should be scheduled to execute. This is achieved by setting environment variable `OMP_PROC_BIND` to `true`. If set to `false`, the place list is ignored and all thread affinity is disabled.

Both these environment variables have a system dependent default value. There is also an interaction between their settings. This is summarized in Figure 4.4.

In case neither of these two variables has been set, thread affinity depends on the system dependent defaults. Affinity may, or may not, be enabled. Given the possible performance impact, we strongly recommend to check the documentation for the defaults, or even better, set them explicitly.

In Figure 4.4, it is seen that if `OMP_PROC_BIND` is set to `false` (and remember this could be the default), thread affinity is *always* disabled. If it is set to `true`, or to one of the supported policies, affinity is enabled and the place list is either the default (again, check the documentation what it is set to), or the list explicitly set through `OMP_PLACES`. If `OMP_PROC_BIND` is not set, but `OMP_PLACES` is explicitly defined, the default setting for `OMP_PROC_BIND` determines whether thread affinity is enabled, or not. This is something to watch out for.

4.5 The OpenMP Places Concept

OpenMP places define where threads are allowed to run. Places are set through environment variable `OMP_PLACES` and fixed throughout the execution of the application.

There are two disjoint ways to define OpenMP places. This section starts with the low level approach, using integer numbers to define where threads can run. Although very powerful, most users do not need this. The second way to specify the places is through the more flexible and elegant abstract names, like "cores," or "threads." This is the preferred method and covered in Sections 4.5.3 and 4.7, starting on page 166 and 188 respectively. The motivation to start with the integer numbers is that they make it easier to introduce and explain some of the features.

4.5.1 Defining OpenMP Places Using Sets with Numbers

An OpenMP *place* consists of an *unordered* set of comma-separated non-negative numbers enclosed by braces ({ and }). An example is the set $\{0, 4, 8, 12\}$ that defines four target resources where a thread could run. Because the order within a place does not matter, the set $\{12, 4, 0, 8\}$ defines the same place.

The specifications use the word "processor" in the context of thread affinity. We prefer to use different terminology, because a processor is too coarse. A contemporary system often has sockets, cores and hardware threads. All of these can be a scheduling unit target for thread affinity.

This is why we take the liberty to introduce our own terminology and refer to the numbers that define a place as *(hardware) resource numbers*, or "resources" for short.

The resource numbers are system and configuration dependent. They are intended to represent an entity the OS can assign work to. Examples are a hardware thread, or a GPU lane.[4]

The way to obtain these numbers for a specific configuration depends on the OS. It is very likely to be obtained by checking a configuration file (for example, file `/proc/cpuinfo` on Linux), through an appropriate OS command (for instance, `numactl` and `lscpu` on Linux, or `psrinfo` on Oracle Solaris), or a specific tool. It is best to check the documentation of the target system and OpenMP compiler for the details.

As an example, Figure 4.5 shows the output of the Linux `lscpu` command on a two socket system using Intel processors. The output shows that this system has two NUMA nodes (these are the two sockets), where each socket has 18 cores with 2 hardware threads per core. This gives a total of $2x18x2 = 72$ hardware threads.

[4]In this chapter we mostly interpret these numbers as hardware thread numbers, but the concept is more general than our usage might suggest.

```
$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                72
On-line CPU(s) list:   0-71
Thread(s) per core:    2
Core(s) per socket:    18
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 79
Model name:            Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
Stepping:              1
CPU MHz:               2995.781
BogoMIPS:              4205.85
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              46080K
NUMA node0 CPU(s):     0-17,36-53
NUMA node1 CPU(s):     18-35,54-71
```

Figure 4.5: **An example of hardware thread numbers** – This is the output from the Linux `lscpu` command on a two socket system. It shows there are two NUMA nodes, the two sockets. This system has a total of $2x18x2 = 72$ hardware threads. The last two lines show the hardware thread numbers and how they map onto the NUMA nodes.

What is most relevant in this output is shown in the last two lines. This shows the hardware thread numbers and how they map onto the two NUMA nodes.

These threads are numbered $0-17$ and $36-53$ on the first NUMA node. The threads on the second NUMA node are numbered $18-35$ and $54-71$. The interpretation requires a basic understanding of the architecture of the system. Numbers $0-17$ represent the first hardware thread on each of the 18 cores on the first socket. The numbers in the $36-53$ range are for the second hardware thread on these cores.

```
$ psrinfo -vp
The physical processor has 16 cores and 128 virtual
processors (0-127)
  The core has 8 virtual processors (0-7)
  The core has 8 virtual processors (8-15)
  The core has 8 virtual processors (16-23)
       ..... output lines omitted .....
  The core has 8 virtual processors (104-111)
  The core has 8 virtual processors (112-119)
  The core has 8 virtual processors (120-127)
    SPARC-T5 (chipid 0, clock 3600 MHz)

The physical processor has 16 cores and 128 virtual
processors (128-255)
  The core has 8 virtual processors (128-135)
  The core has 8 virtual processors (136-143)
  The core has 8 virtual processors (144-151)
       ..... output lines omitted .....
  The core has 8 virtual processors (232-239)
  The core has 8 virtual processors (240-247)
  The core has 8 virtual processors (248-255)
    SPARC-T5 (chipid 1, clock 3600 MHz)
```

Figure 4.6: **Another example of hardware thread numbers** – This is an output fragment of the Solaris `psrinfo -vp` command on a SPARC system. A virtual processor refers to what is called a hardware thread in this book. The output shows there are two sockets with 16 cores each. Each core has 8 hardware threads. This gives a total of 256 hardware threads for this system.

As an example, the first core on the first NUMA node has hardware threads 0 and 36. These numbers are 18 and 54 for the second NUMA node.

When using the integer numbers, this type of output is indispensable to define the places. For example, the place set $\{0, 1, 2, 18, 19, 20\}$ defines the first three hardware threads on the first three cores of both NUMA nodes.

To demonstrate that the numbering scheme depends on the hardware and OS, the output from the `psrinfo -vp` Solaris command on a two socket system using SPARC processors is given in Figure 4.6. For formatting reasons, several lines have been omitted and the layout has been somewhat adapted.

```
<lower-bound> : <count> [:  <stride> ]
```

Figure 4.7: **Syntax of the interval notation to define an OpenMP place** – The `lower-bound` and `count` numbers are positive integers, including zero. The optional `stride` can be any number, including zero, or a negative value. The default is 1.

The output shows this system has two sockets. Each socket has 16 cores with 8 hardware threads ("virtual processor") per core. This means there are $16x8 = 128$ hardware threads per socket. The threads are numbered $0 - 127$ on the first socket. The threads on the second socket are numbered $128 - 255$. On this system, place set $\{0, 1, 2, 8, 9, 10\}$ also defines the first three hardware threads on the first two cores of the first socket, but it is a different set than the one used on the other system.

There is an important thing to remember. The numbers within a single place are *unordered*. From a scheduling point of view, there is no preference for entries within the set. This allows the user to provide flexibility to the OpenMP runtime scheduler. An example may be the hardware threads that are part of a core. There shouldn't be any difference on which hardware thread the OpenMP thread is running.

This also allows to optimize for the situation that the memory access time is uniform for a certain subset of the hardware threads. If an OpenMP place is created that includes all these threads, the scheduler has the freedom to select an arbitrary thread from this set, without an impact on the performance.

Although OpenMP places are designed to express uniformity in the architecture, this is not required and obviously not verified by the runtime system. It is the responsibility of the user to define places in a meaningful way. If for example the memory access time is not uniform for all hardware threads within a single place, performance variations across identical runs may, and probably will, be observed.

The output in Figures 4.5 and 4.6 points to a potential issue: the list with numbers could get rather long. This is not only tedious to type in, but also error-prone. This is the reason that a convenient interval notation is supported. An interval is specified through a lower bound, a count[5], and a stride. The syntax of the interval notation is given in Figure 4.7.

As an example, the following list defines all 36 hardware threads spanning the first socket of the Intel system: $\{0 : 18 : 1, 36 : 18 : 1\}$. This compact notation expands to $\{0, 1, \ldots, 17, 36, 37, \ldots, 53\}$. On the SPARC system, to include the 64

[5]Note this is a count, not the maximum.

hardware threads on the first 8 cores of the second socket, the list must be set to $\{128:64:1\}$.

There are three things worth mentioning in relation to the interval notation:

- The stride can be negative, and although not very meaningful, even be zero.
- The interval notation can be used in combination with comma separated lists.
- The exclusion operator "!" can be used to exclude the number or places immediately following the operator.

4.5.2 The OpenMP Place List

In most cases, a single place is not sufficient and an *OpenMP place list* must be constructed. A place list is an *ordered* list of places.

It is important to note that, while the order of the resource numbers within an individual place is irrelevant, the order of the places in the place list matters, As an example, consider the place list defined by "$\{0,8\},\{128,136\}$." The place list "$\{8,0\},\{136,128\}$" is equivalent, but list "$\{128,136\},\{0,8\}$" is different. Both choices regarding the ordering (or lack thereof) make sense.

Multiple resource numbers within a single place give the scheduler more freedom to select a target location where to run the thread. As all resource numbers within one place are considered to be equivalent, the order does not matter.[6]

This is different for the place list. The order of the places within the place list affects the selection of places for a specific affinity policy. The same policy with a permuted place list may result in a different mapping of threads onto the hardware. The interpretation of the place list under a given affinity policy is extensively covered in Section 4.6, starting on page 168.

The interval notation may also be applied to an entire place. This provides a convenient way to define long place lists through a compact notation. The entire place definition is used as the lower bound. The stride is added to each of the resource numbers in this place.

As an example, consider this place list: $\{0,1,2,3\}:3:5$. By definition of an interval, this means there are 3 copies, each separated by a stride of 5. The resulting expanded list consist of the following places: $\{0,1,2,3\}$, $\{0+1*5,1+1*5,2+1*$

[6] In case this kind of freedom for the runtime scheduler is not desirable, the solution is to create places consisting of one element only.

$5, 3 + 1 * 5\}$, $\{0 + 2 * 5, 1 + 2 * 5, 2 + 2 * 5, 3 + 2 * 5\}$. In other words, the resulting place list is the following: $\{0, 1, 2, 3\}, \{5, 6, 7, 8\}, \{10, 11, 12, 13\}$. This may even be written in a more compact way, because the base place may also be defined as $\{0 : 4 : 1\}$. Using this notation, the same place list is defined by $\{0 : 4 : 1\} : 3 : 5$.

4.5.3 Defining OpenMP Places Using Abstract Names

Specific resource numbers in places, and the place list, provide full control where threads should run, but portability of the definitions is not guaranteed. On a different system, or even a different configuration of the same system, the numbers most likely need to be adjusted. This is why the following three *abstract names* are supported as an alternative:

- `sockets` - Each place corresponds to a single socket. A socket can have multiple cores.
- `cores` - Each place corresponds to a core. A core can support multiple hardware threads.
- `threads` - Each place corresponds to a single hardware thread.

Although the definition of these abstract names is system dependent, they can be expected to be meaningful and reflective of the underlying architecture. This is why they are recommended over the resource numbers. The abstract names and resource numbers are mutually exclusive. In other words, an abstract name can *not* be combined with a list specification.

An implementation can add abstract names to the above list. The usual pros and cons apply. Most likely, these names better support the target architecture, but portability is lost.

The abstract names are hierarchical. A socket typically has multiple cores and each core can have multiple hardware threads. As an example, the `OMP_PLACES=sockets` place list definition implies that the OpenMP threads may run on any of the hardware threads associated with a socket. There is no distinction as to which cores are used for example.

Transparent to the user, an abstract name defines a place list. The places in this list span the resource specified. Each place can have more than one resource number in case the name used has multiple hardware scheduling units associated with it. For example, the `OMP_PLACES=cores` definition internally uses a place

list that consists of a set of places. Each place contains the resource number(s) corresponding to a core. If a core supports eight hardware threads, say, each place consists of a set with eight unique numbers.

The abstract name can be appended with an optional number between parentheses. This specifies the length of the place list that is created. For example a place list defined by `OMP_PLACES=cores(4)`, restricts the list to only four cores. It is not possible to specify which cores. They may be on the same socket, or not. If this level of detail is needed, resource numbers must be used.

In case the length of the list exceeds the number of resources (of the named type) available, the length of the generated place list is also implementation dependent.

The place list is only half of the solution. The affinity policy dictates in what way the places are mapped onto the hardware. A much more extensive coverage of the abstract names, in combination with the affinity policy, may be found in Section 4.7, on page 188.

4.5.4 How to Define the OpenMP Place List

The place list is defined through the `OMP_PLACES` environment variable. If the user does not set this variable to define the place list explicitly, the system uses an implementation-dependent list.

An example how to set the place list is given below. For demonstration purposes, the list based notation is combined with the interval notation:

`OMP_PLACES="{0,8},{128,136},{112:8},{240:8}"`. This list consists of four places. The first two places contain two resource numbers each, while the remaining two places each consist of eight resource numbers. The abstract names are set through the same environment variable. For example: `OMP_PLACES="cores(4)"`.

The places and the place list definitions are *static*. There are no runtime functions to change the place list while the program is executing. In case the place list contains resources that are not available, for example a non-existing hardware thread number, the behavior is implementation-dependent. The runtime system may ignore it and use a default place list, or chose to abort with an appropriate error message. More information on this environment variable can be found on page 58 in Section 2.2.

4.6 Mapping Threads onto OpenMP Places

The next question to answer is how the places are used to map the OpenMP threads onto the hardware. This is controlled through an interaction between the place list and the thread affinity policy. More specifically, the selected affinity policy is applied at the level of the places in the place list.

Implied with thread affinity is "thread binding" at the level of an OpenMP place. This means that the threads within the same team are not allowed to move out of the place they were originally assigned to.

Once the destination place to run on has been determined, the OpenMP thread is not allowed to execute elsewhere. The thread may be temporarily de-scheduled to make room for another activity, but once it is scheduled to run again, it must be put back onto the same place. If the place has multiple resource numbers (e.g. {0,1,2}), the thread is allowed to be scheduled to run on a different resource number, as long as it is within the same place.

Affinity and binding are controlled through environment variable `OMP_PROC_BIND`. This variable defines the policy, or policies in case of nested parallelism, applied to parallel regions.

- `OMP_PROC_BIND=false` - All thread affinity and binding is disabled. Any place list, set through environment variable `OMP_PLACES` is ignored. The same is true for usage of the `proc_bind` clause on the parallel region(s).

- `OMP_PROC_BIND=true` - Thread binding is enabled and the first thread is bound to the first place in the place list. No other guarantees regarding binding and placement are given.

  If the first thread cannot be bound, the behavior is implementation dependent.

  This is a legacy setting, available prior to OpenMP 4.0. We do not suggest to use it and specify a policy, or multiple policies, instead.

- `OMP_PROC_BIND=<comma separated list with policies>` - At each nesting level, the threads are placed according to the affinity policy in effect at that level. Threads are not allowed to migrate between the places.

As with all OpenMP environment variables, this variable defines the initial setting(s). At runtime an already defined policy may be overruled, or defined if not yet set, through the `proc_bind` clause supported on the `parallel` construct.

#pragma omp parallel proc_bind(*policy***)** *[clause[[,] clause]...]* *new-line* *structured block*
!$omp parallel proc_bind(*policy***)** *[clause[[,] clause]...]* *structured block* **!$omp end parallel**

Figure 4.8: **Syntax of the proc_bind clause on the parallel construct in C/C++ and Fortran** – The thread affinity policy may be specified at each nesting level. It is only valid for the region it applies to. Choices for the policy are `master`, `close`, or `spread`.

This is a straightforward clause, taking one of the three affinity policies as an argument. The syntax in C/C++ and Fortran is given in Figure 4.8.

The affinity policy, as set through the `proc_bind` clause, is specified on each parallel region and not persistent. It is only valid for the duration of the particular region it applies to. In absence of the `proc_bind` clause, the corresponding setting through `OMP_PROC_BIND` takes effect. Be aware that the default setting for variable `OMP_PROC_BIND` is implementation dependent. In addition to define `OMP_PLACES` explicitly, it is recommended to set this variable as well.[7]

The policy defines in what way OpenMP threads are mapped onto the hardware threads. It can be adapted on every parallel region, but the setting is static for the duration of the execution of the parallel region it applies to. The supported policies are summarized in Figure 4.9.

Irrespective of the policy, once an OpenMP thread has been assigned to a place, it is not allowed to move to another place. This is called "thread binding."

To simplify the description of the affinity algorithms and the mappings, the notation defined in Figure 4.10 is used in the remainder of this chapter.

In all examples to follow, the generic multi-core architecture shown in Figure 4.1 on page 152 is the target system. The topology of this system can now be described in terms of OpenMP places. This system is relatively simple and the abstract names are sufficient, but to demonstrate the concepts, the resource numbers are used instead.

The system has two sockets with four cores each. A core supports four hardware threads. The threads within a core are fully symmetric in terms of memory access

[7] To easily verify the settings, environment variable `OMP_DISPLAY_ENV` is recommended to be set to `true`, or `verbose`.

Policy	Description
`master`	Each thread in the team is assigned to the same place as the master thread.
`close`	The threads in the team are placed close to the master thread. Threads are assigned to consecutive places in a round-robin way, starting with the place to the right of the master thread.
`spread`	Spread the threads as evenly as possible over the places.

Figure 4.9: **Supported policies for thread binding** – A specific policy is applied to a parallel region. In the case of nested parallelism, each parallel region has its own policy. The policy may be set through environment variable `OMP_PROC_BIND`, by using the `proc_bind` clause on a parallel region, or both. In case of the latter, the clause overrules the environment variable.

Symbol	Description
P	The number of places.
P_i	Place "i."
SP_i	Subpartition "i."
T	The number of threads in a team.
T_i	A thread with thread number "i" within a team.
ST_i	A subset of threads in a team, referred to with identifier "i."
$T_i \in P_j$	OpenMP thread "i" is scheduled to execute in place P_j.
$T_i \in SP_j$	OpenMP thread "i" is scheduled to execute in partition SP_j.

Figure 4.10: **Notation used in the affinity policy descriptions** – In the description of the various thread affinity policies the above notation is used. Places and subpartitions are considered as sets. This is why the mathematical $\in$ symbol is used to denote a thread is assigned to a place, or subpartition.

times. Memory access within a socket is uniform. In this respect, all cores within a socket are equal, but at the cache level there could be a difference in access times. From a memory access point of view, this system has four places ($P = 4$), symbolized by $P_0, \ldots, P_3$. Using the numbering for the hardware threads shown in Figure 4.1, the definition of each place is listed in Figure 4.11, where each place is defined to consist of the four hardware threads that are part of a core.

The place list consist of the four places $\{P_0, \ldots, P_3\}$ and is defined through environment variable `OMP_PLACES` as follows:

```
OMP_PLACES = "{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}".
```

	Definition	
Place	**Full List**	**Interval Notation**
P_0	$\{0, 1, 2, 3\}$	$\{0 : 4 : 1\}$
P_1	$\{4, 5, 6, 7\}$	$\{4 : 4 : 1\}$
P_2	$\{8, 9, 10, 11\}$	$\{8 : 4 : 1\}$
P_3	$\{12, 13, 14, 15\}$	$\{12 : 4 : 1\}$

Figure 4.11: **Place definitions used in the examples in this section** – In the examples that illustrate the various affinity policies, the above definitions for the places are used. Both the full list expansions, as well as the equivalent interval notations are shown.

This may also be written in a more compact way, using the interval notation: `OMP_PLACES = "{0:4:1},{4:4:1},{8:4:1},{12:4:1}"`. Recognizing the regularity in this sequence, the definition may even be further condensed to `OMP_PLACES = "{0:4:1}:4:4"`. In the remaining subsections it is shown in what way this place list, in combination with the various affinity policies, is used to control where the OpenMP threads execute.

Before we go into the details, there are several comments to be made. They apply to all the placement policies and examples discussed below.

- The mapping of OpenMP threads onto the hardware is decided at the level of *places*. If a place consists of multiple resource numbers, the selection of a specific resource within the place depends upon the implementation.

- The phrase "a thread executes in a place" is a shortcut for the following scenario. The OpenMP runtime system selects an OpenMP thread to be scheduled onto the place. Next, the runtime scheduler selects a resource number within the place, to execute this thread.

- According to the specifications, the *initial placement* of the master thread of a team is inherited from the place where the parent thread of this master thread is running.

  There is a potential catch with this. Upon program start up, the OS decides on the placement of the process. At this point, the OpenMP runtime system is not involved and there is no concept of OpenMP places yet. Therefore, it may happen that the OS schedules the parent thread to run on a hardware resource that is not included in any of the places.

In case thread migration is supported on the system, the OpenMP runtime system first migrates the parent thread to a place in the place list and decides on the placement of the master thread next. If migration is not supported however, or the affinity request cannot be fulfilled for some other reason, the behavior is implementation-dependent.

We recommend to check the documentation of the specific runtime system used to verify the behavior in such cases.

- In the remainder, another shortcut is used. Whenever the place of the master thread is specified, it is implied that the selection of this place has been derived from the place where the parent of this master thread executes.

- To simplify the drawings, unless noted otherwise, in all examples, the parent of the master thread is assumed to execute in place P_0 defined in Figure 4.11.

4.6.1 The Master Affinity Policy

The most straightforward affinity policy is the `master` policy, where all threads in the team are scheduled to run in the same place the master thread executes in.

This policy is useful when efficient communication and data sharing between threads is important. This is under the assumption that the memory bandwidth requirements of the threads are modest. Otherwise, this policy may cause a bottleneck if all threads execute on the same socket, or core.

The `master` affinity policy is illustrated in Figure 4.12. The threads are symbolized by solid circles. The master thread has been colored black, while the gray circles represent the other threads in the team. If the master thread executes in the second place of the place list, $P_1 = \{4, 5, 6, 7\}$, the other threads in the team run there as well. Effectively, all threads in this team execute on the same core.

Because there are four threads in total, and four resources in this place, each OpenMP thread may execute on a different hardware thread, but this is not guaranteed.

The specifications require only that the other threads in the team to execute in the same place. Where they execute within this place, is an implementation-dependent choice. A good scheduler will exploit the presence of additional hardware threads and does not overload resources if this can be avoided.

If the number of OpenMP threads exceeds the number of resources within the place, multiple threads must share a resource. If there are eight threads in our

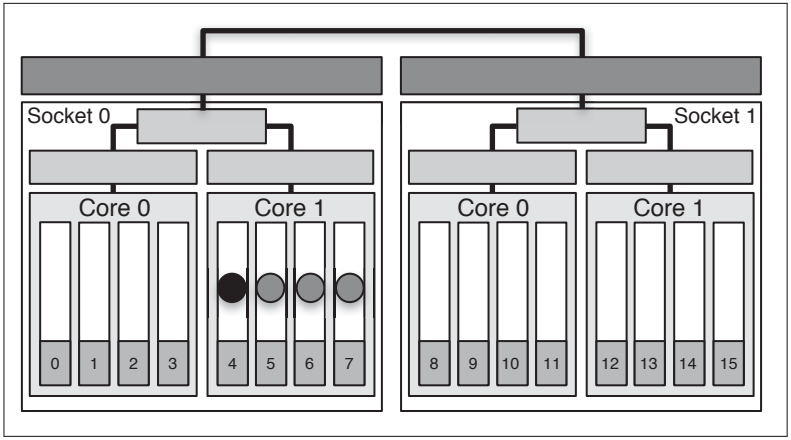

Figure 4.12: **An example of the master affinity policy** – With this policy, additional threads of the same team are scheduled to run in the same place where the master thread is executing. Threads are symbolized by solid circles. The master thread has a black color. The gray circles represent the other threads in the team. Because the master is assumed to execute in place P_1, the other threads execute in the same core.

example, each hardware thread in core 1 of socket 0 executes two OpenMP threads. This is probably not a good approach, because oversubscription of hardware threads usually degrades performance.

4.6.2 The Close Affinity Policy

The goal of the `close` affinity policy is to keep OpenMP threads close enough to take advantage of data locality, while still spreading the threads across the system as much as possible.

In a typical scenario, the cores (and hardware threads) within a single socket are used. Combined with the First Touch page placement policy, the data is located in the memory connected to the socket, and more expensive remote memory accesses are avoided.

Using the notation from Figure 4.10, the number of threads is denoted by T, and the number of places is symbolized by P. There are two cases to distinguish:

- If $T \leq P$, then there are sufficient places and each thread is assigned to a unique place.

Threads are selected by their thread number within the team, starting with the lowest numbered thread, and assigned to *consecutive* places in the place list. The first place selected is the one where the parent of the master thread executes.

For example, if $T = 3$ and $P = 4$, there are four places $P_0, \ldots, P_3$. Assuming the parent of thread T_0 executes in P_0, thread T_0 is assigned to place P_0 and the other threads T_i execute in place P_i, for $i = 1, \ldots, 3$.

- If $T > P$, then there are more threads than places and at least one place executes more than one OpenMP thread.

  Each place gets assigned a *consecutive* subset of the total number of threads T in the team. There are P subsets and the number of threads ST_i in the subset is chosen, such that $floor(T/P) \leq ST_i \leq ceiling(T/P)$, $i = 0, \ldots P - 1$.[8]

  The first subset includes the master thread and is assigned to the place where the parent of the master thread is running. The second subset is assigned to the place next to this place in the place list, and so on.

  As an example, consider $T = 8$ and $P = 4$. Each subset contains $floor(8/4) = ceiling(8/4) = 2$ threads. Assuming the parent of the master thread executes in place P_0, threads T_0 and T_1 are assigned to P_0. Threads T_2 and T_3 are assigned to place P_1, and so on.

  If P does not divide T evenly, the lower and upper bounds for ST_i differ by one and not all subsets contain the same number of threads. It is up to the implementation to assign subsets to places. In other words, at least one place gets assigned one more thread than other places and the implementation decides which place(s) to select for this.

  Consider a case with seven threads ($T = 7$) and four places ($P = 4$). There is one subset with a single thread and three subsets have two threads each.[9] It is up to the implementation how to map the subsets onto the places. For example P_1, P_2 and P_3 may all execute two threads each, while the first place P_0 gets assigned one thread only.

[8] The Glossary contains the definitions of these functions.

[9] For the interested reader, "Computing the number of threads per subset" in the Glossary explains how these results may be obtained.

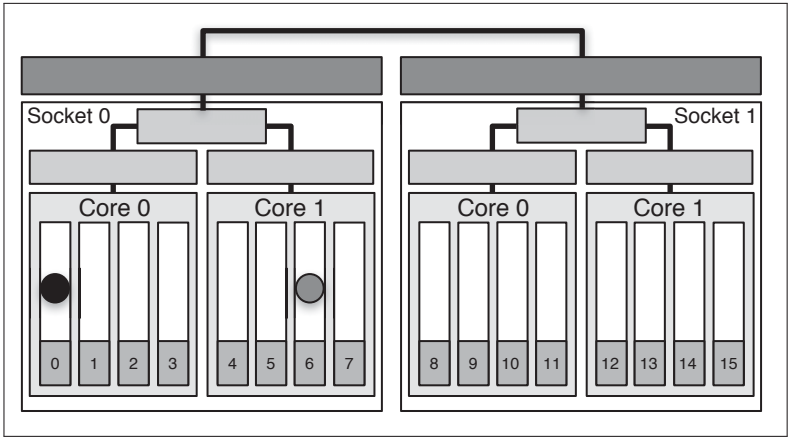

Figure 4.13: **An example of the close affinity policy using 2 threads** – With this policy, additional threads of the same team are scheduled close to the place where the master thread is executing. Threads are symbolized by solid circles. The master thread has a black color. The gray circle represents the other thread in the team. With the given place list, and assuming the master thread executes on core 0 of socket 0, the second thread executes on core 1 on the same socket.

The `close` affinity policy is illustrated in Figure 4.13 for a case with four places ($P = 4$) and two threads ($T = 2$). The $T \leq P$ algorithm applies.

The assumption is that the master thread executes in the place (P_0). This means the second thread is scheduled on a hardware thread from place P_1. Because each place spans a core, the threads effectively run on separate cores. In this case, both cores used are on the first socket, but if the parent of the master thread executes in place P_1, for example, the second thread executes on core 0 of the second socket.

The choice of a resource number within a specific place is determined by the runtime scheduler and the order within a single place is assumed to be irrelevant regarding scheduling preferences. This is illustrated in Figure 4.13. In this case, hardware threads 0 and 6 are selected.

If we increase the number of threads to three, the third place in the place list is added as a scheduling target and this third thread executes in core 0 on socket 1 (place P_2). Again the choice of a hardware thread within this core is determined by the scheduler. With four threads, core 1 on the second socket is also used.

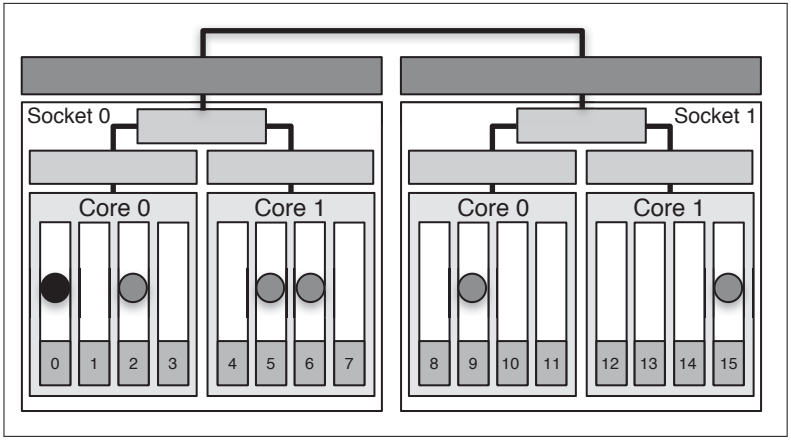

Figure 4.14: **An example of the close affinity policy using 6 threads** – The master thread executes on core 0 of socket 0, that is, in place P_0. Four threads are scheduled onto the four cores. The implementation selects the places for the remaining threads. These are assumed to be P_0 and P_1 in this example.

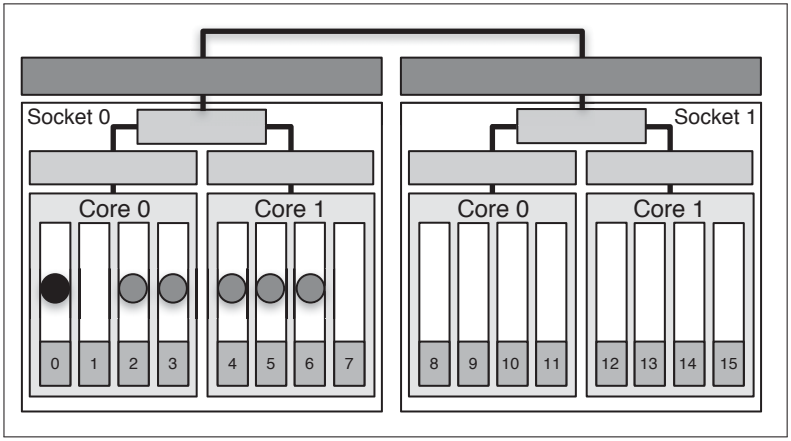

Figure 4.15: **Using the close affinity policy to force 6 threads onto 1 socket** – With a modified place list spanning the first two cores only, the scheduler uses the two cores on the first socket. Because there are two places, the six threads are balanced across the two cores.

Once the number of threads exceeds the number of places in the place list ($T > P$), multiple threads are scheduled onto the same place. The details of the number of threads per place is implementation dependent. For example, if six threads are used, $floor(6/4) = 1$, $ceiling(6/4) = 2$ and two of the four places get assigned two threads. It is up to the implementation which places to select. A possible choice is shown in Figure 4.14.

This example illustrates another point. Although we used the `close` affinity policy, the threads are still spread over the entire system. This is because the place list spans all resources in the system.

When spreading the threads over the system is not desired, the solution is to modify the place list. As an example, assume all threads need to execute within the first socket only, but they should be balanced across the two cores. This is achieved by re-defining the place list to include only places P_0 and P_1. The `OMP_PLACES` environment variable must be set as follows: `OMP_PLACES = "{0,1,2,3},{4,5,6,7}"`. Alternatively, with the more compact interval notation, the same place list is given by `OMP_PLACES = "{0:4:1},{4:4:1}"`, or even shorter: `OMP_PLACES = "{0:4:1}:2:4"`. With this new place list, threads are assigned to only the two cores in the first socket. The assignment algorithm ensures load balancing across the two cores.

Figure 4.15 illustrates a case in which there are six threads. Because $T = 6$ and $P = 2$, the $T > P$ algorithm applies. There are two subsets and both contain $floor(6/2) = ceiling(6/2) = 3$ threads. Each subset gets assigned to a place.

As before, the mapping of OpenMP threads within a single place depends on the scheduler. It is guaranteed that only the first socket is used and each core executes three threads that are consecutively numbered.

If the parent of the master thread executes in place P_0, threads 0, 1 and 2 execute on core 0, while the three remaining threads 3, 4, and 5 execute in place P_1. This means they run on core 1.

As before, the selection of hardware threads within both places is up to the scheduler. It is a quality of implementation issue which hardware threads are selected. In this case, the ideal scenario is that all OpenMP threads execute on a different hardware thread.

4.6.3 The Spread Affinity Policy

The third policy, places the OpenMP threads as far apart as possible. As before, this is in the context of places.

One of the situations in which the `spread` policy can be of benefit, is in the case of an application that is memory-bandwidth demanding. The bandwidth provided by the various memories across the sockets, is best leveraged by distributing the threads over the sockets.[10]

Transparent to the user, the `spread` policy achieves its goal by splitting the place list into *subpartitions*. The OpenMP threads are mapped onto subpartitions, such that the distance between them is maximized.

The algorithm to define the subpartitions depends on the number of threads T and the number of places P in the place list:

- $T \leq P$ - The place list is split into T subpartitions. Each subpartition contains at least $floor(P/T)$, and at most $ceiling(P/T)$ *consecutive* places.

  Because there are T subpartitions, each thread is assigned to a subpartition, but a subpartition may consist of more than one place.

  Thread assignment starts with the master thread. It executes in the subpartition that contains the place where the parent thread runs. The thread with the next smallest thread number is assigned to the first place in the next subpartition, and so forth for subsequent threads. Relative to the starting point, the assignment wraps around.

  For example, if $T = 4$ and $P = 8$, there are four subpartitions $SP_0, \ldots, SP_3$ and each subpartition consists of $floor(8/4) = 2$ consecutive places.

  Using the notation from Figure 4.10, and assuming the parent of the master thread runs in place P_3, the subpartition definitions and thread assignments are described as follows:

  $$SP_0 = \{P_0, P_1\} \text{ and } T_3 \in P_0$$
  $$SP_1 = \{P_2, P_3\} \text{ and } T_0 \in P_2$$
  $$SP_2 = \{P_4, P_5\} \text{ and } T_1 \in P_4$$
  $$SP_3 = \{P_6, P_7\} \text{ and } T_2 \in P_6$$

[10] This of course assumes the data is also spread out over the various memories.

The master thread (T_0) is assigned to subpartition SP_1 and because the first place is used first, it executes in place P_2. As shown, subsequent threads are mapped onto the subsequent places, wrapping around. This is why thread T_3 executes in P_0.

There are four threads only and half of the eight places are used. When there are eight threads ($T = 8$), each subpartition consists of a single place: $SP_i = P_i$ for $i = 0, \ldots, 7$. In other words, each thread is assigned to a unique place.

When T does not divide P evenly ($floor(P/T) \neq ceiling(P/T)$), not all subpartitions contain the same number of places. All subpartitions have at least $floor(P/T)$ places, but at least one subpartition includes an additional place. The decision of which subpartition includes an additional place is implementation-dependent.

As an example, if $T = 5$ and $P = 8$, there are five subpartitions. Out of these, three contain $ceiling(8/5) = 2$ places and two consist of a single place each ($floor(8/5) = 1$). In such a case, an obvious choice is to assign places in a round-robin manner. In the scenario above, the first three threads may execute in a subpartition with two places, while the last two threads run in a subpartition with a single place.

If the parent of the master thread executes in place P_3 again, the following is a possible distribution of the threads over the subpartitions and places:

$SP_0 = \{P_0, P_1\}$ and $T_4 \in P_0$
$SP_1 = \{P_2, P_3\}$ and $T_0 \in P_2$
$SP_2 = \{P_4, P_5\}$ and $T_1 \in P_4$
$SP_3 = \{P_6\}$ and $T_2 \in P_6$
$SP_4 = \{P_7\}$ and $T_3 \in P_7$

Clearly, not all places are used. Such "leftover" places may be used in the case of nested parallelism.

- $T > P$ - The place list is split into P subpartitions. All subpartitions contain a single place and ST_i *consecutively* numbered threads are assigned to each subpartition. The number of threads in the subset is chosen, such that $floor(T/P) \leq ST_i \leq ceiling(T/P)$.

There are more threads than places and at least one place has more than one thread assigned to it.

Thread assignment to subpartitions is as follows. The first subset with ST_0 threads includes the master thread. It is assigned to the subpartition containing the place of the parent of the master thread. The next subset with ST_1 threads, is assigned to the next subpartition, and so on.

For example, if $T = 8$ and $P = 4$, there are four subpartitions $SP_0, \ldots, SP_3$. Each has $floor(8/4) = ceiling(8/4) = 2$ consecutive threads assigned to it.

If it is again assumed that the parent of the master thread executes in place P_3, the assignment of threads to places is as follows:

$SP_0 = \{P_0\}$ and $T_2, T_3 \in P_0$
$SP_1 = \{P_1\}$ and $T_4, T_5 \in P_1$
$SP_2 = \{P_2\}$ and $T_6, T_7 \in P_2$
$SP_3 = \{P_3\}$ and $T_0, T_1 \in P_3$

If P does not divide T evenly, the number of threads per subpartition is implementation-dependent.

Let's assume $T = 7$ and $P = 4$. In this case there are again 4 subpartitions and each consists of a single place. Because $floor(7/4) = 1$ and $ceiling(7/4) = 2$, two threads are assigned to three partitions, while one partition has a single thread assigned to it.

It is up to the implementation to decide how many threads there are in each subpartition. In the example below, only one thread is assigned to partition P_2. All other partitions have two threads:

$SP_0 = \{P_0\}$ and $T_2, T_3 \in P_0$
$SP_1 = \{P_1\}$ and $T_4, T_5 \in P_1$
$SP_2 = \{P_2\}$ and $T_6 \in P_2$
$SP_3 = \{P_3\}$ and $T_0, T_1 \in P_3$

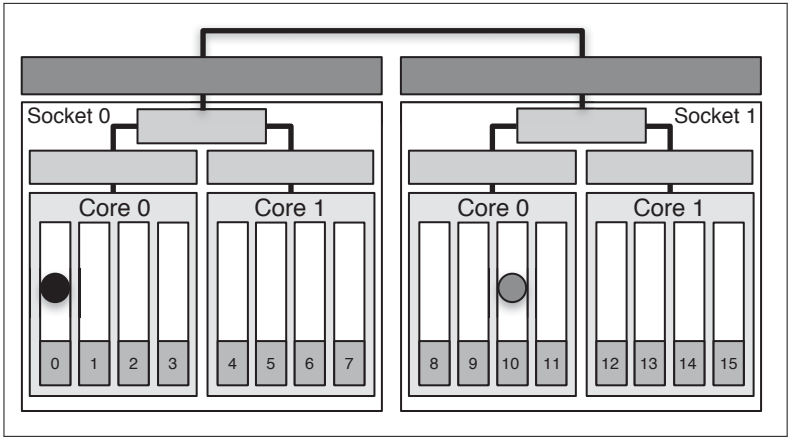

Figure 4.16: **An example of the spread affinity policy using 2 threads** – If the master thread executes on core 0 of socket 0, the second thread is scheduled to execute on core 0 of the other socket.

Before we continue, some more terminology needs to be explained. In the specifications, a subpartition is also referred to as a *place partition*. It is defined as an ordered list that corresponds to a contiguous list in the place list. In this book, we consistently use the word *subpartition* though.

An example of the `spread` affinity policy is illustrated in Figure 4.16. There are two threads ($T = 2$) and the place list from Figure 4.11 is used ($P = 4$).

There are fewer threads than places and the "$T \leq P$" algorithm applies. Two subpartitions, SP_0 and SP_1, are created. Both contain $floor(4/2) = 2$ consecutive places: $SP_0 = \{P_0, P_1\}$ and $SP_1 = \{P_2, P_3\}$. Assuming that the parent of the master thread executes in either place P_0, or P_1, the master thread is assigned to P_0 in subpartition SP_0. The second thread is scheduled onto place P_2, because this is the first place in the second subpartition SP_1. Given the definition of the place list, the two threads are scheduled on the first core of each socket, effectively spreading them over the two sockets.

If four threads are used, four subpartitions are created. Each of these spans the four threads in a different core, and a thread is assigned to one of the subpartitions. The result is that the four threads are spread over the four cores in the system. This is shown in Figure 4.17. As before, the selection of a resource number within a place is system dependent.

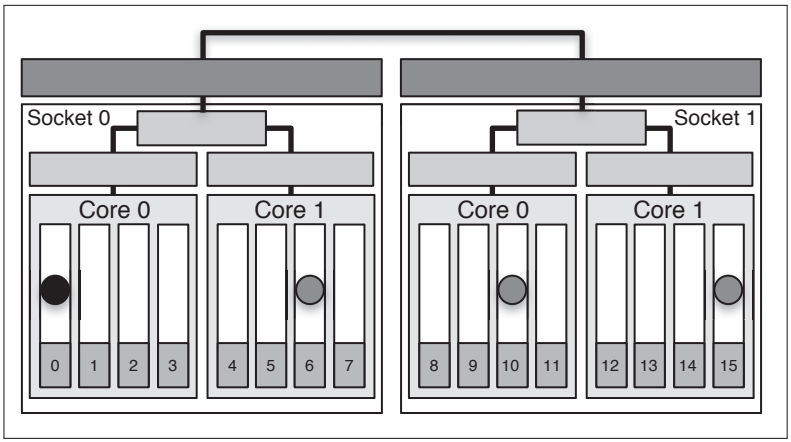

Figure 4.17: **An example of the spread affinity policy using 4 threads** – In this case, each core executes one thread. The mapping of the OpenMP threads onto the hardware threads within a core is implementation dependent.

Things get interesting if more than four threads are used. As an example, consider a scenario with six threads. This means that $T = 6$ and $P = 4$. The "$T > P$" algorithm applies and a total of four subpartitions are created. Each consists of a single place: $SP_i = P_i$ for $i = 0, \ldots, 3$. There are six threads to be distributed over these four subpartitions. Two of these subpartitions have two threads assigned to them, while the other two subpartitions contain one thread only. It is assumed that the parent of the master thread executes in place P_0. A possible assignment may be as follows:

$$SP_0 = \{P_0\} \text{ and } T_0, T_1 \in P_0$$
$$SP_1 = \{P_1\} \text{ and } T_2 \in P_1$$
$$SP_2 = \{P_2\} \text{ and } T_3, T_4 \in P_2$$
$$SP_3 = \{P_3\} \text{ and } T_5 \in P_3$$

Given the definitions and architecture, this is a very good placement. There is, however, no guarantee that this is what the runtime system does. How can it know what is best? It has visibility at the places level only and no understanding of the underlying architecture.

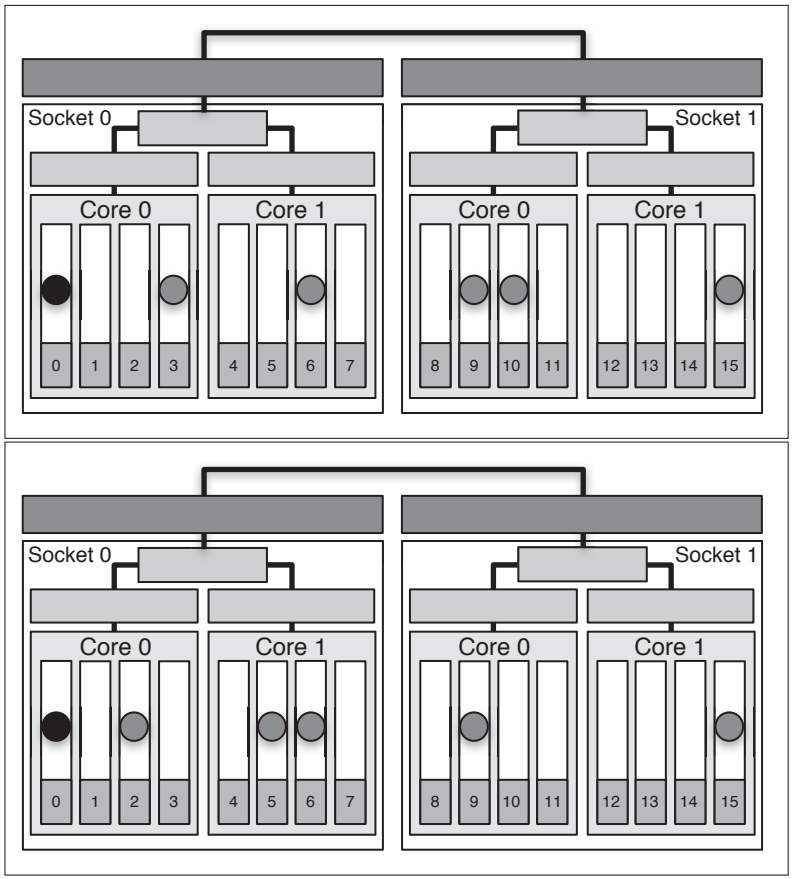

Figure 4.18: **Placement examples using the spread affinity policy with 6 threads** – In the first example, the placement is balanced and optimal, but there is no guarantee this is what the runtime system does. With the given place list, the second, unbalanced, distribution is equally valid.

The assignment below is equally valid for example, but now four threads execute within one socket, while only two threads execute in the other socket:

$$SP_0 = \{P_0\} \text{ and } T_0, T_1 \in P_0$$
$$SP_1 = \{P_1\} \text{ and } T_2, T_3 \in P_1$$

$SP_2 = \{P_2\}$ and $T_4 \in P_2$

$SP_3 = \{P_3\}$ and $T_5 \in P_3$

Both mappings are illustrated in Figure 4.18. As before, the scheduling of threads within one place is implementation dependent.

If this variation in the mapping is undesirable, the solution is to redefine the place list and eliminate undesired implementation-dependent choices. The most elegant solution is to define two places. Each place spans a socket:

$P_0 = \{0 : 8 : 1\} = \{0, \ldots, 7\}$

$P_1 = \{8 : 8 : 1\} = \{8, \ldots, 15\}$

The place list is defined as $\{P_0, P_1\}$. In this case, $T > P$ and there are two subpartitions. Each subpartition consists of a single place with $floor(6/2) = 3$ threads assigned to it. Assuming the parent of the master thread executes in P_0, the assignment is as follows:

$SP_0 = \{P_0\}$ and $T_0, T_1, T_2 \in P_0$

$SP_1 = \{P_1\}$ and $T_3, T_4, T_5 \in P_1$

This achieves load balancing over the two sockets with the guarantee that both sockets get assigned three threads each. The one uncertainty is whether the scheduler spreads these three threads over the two cores in each socket, or places all three in only one of the two cores on each socket.

The performance of the latter distribution is most likely sub-optimal, because the caches and bandwidths on the socket are not fully utilized. To avoid this scenario, a more complex place list must be constructed. The idea is to create more places in the place list. Define the following six places:

$P_0 = \{0, 1\}$

$P_1 = \{2, 3\}$

$P_2 = \{4, 5, 6, 7\}$

$P_3 = \{8, 9\}$

$P_4 = \{10, 11\}$

$P_5 = \{12, 13, 14, 15\}$

The place list is modified and consists of these six places, listed in the following order: $\{P_0, P_1, P_2,\ P_3, P_4, P_5\}$. With this new definition, the $T \leq P$ algorithm applies. There are six subpartitions and each consists of a single place only. Assuming that the parent of the master thread executes in place P_2, the mapping of threads onto places is as follows:

$$SP_0 = \{P_0\} \text{ and } T_4 \in P_0 = \{0, 1\}$$
$$SP_1 = \{P_1\} \text{ and } T_5 \in P_1 = \{2, 3\}$$
$$SP_2 = \{P_2\} \text{ and } T_0 \in P_2 = \{4, 5, 6, 7\}$$
$$SP_3 = \{P_3\} \text{ and } T_1 \in P_3 = \{8, 9\}$$
$$SP_4 = \{P_4\} \text{ and } T_2 \in P_4 = \{10, 11\}$$
$$SP_5 = \{P_5\} \text{ and } T_3 \in P_5 = \{12, 13, 14, 15\}$$

With these definitions, there are three threads per socket. It is guaranteed that one of the cores on each socket executes two OpenMP threads, while the other core on the same socket executes a single thread.

This is illustrated in Figure 4.19. In this example, it is assumed that the parent of the master thread executes in place P_2. As may easily be verified, regardless of where the parent of the master thread executes, the number of threads per core is always the same as above.

The places could have been defined differently to achieve a similar distribution. The two sockets, as well as the two cores within each socket, are symmetric. Within a socket, the number of threads per core could be interchanged, without an impact on the performance. All that is required is a slight change in the definition of the places.

Although any order of the places in the place list above achieves the same goal, there is a reason the place list is defined in the order shown here. If there are only two threads, they are scheduled onto a different socket. To be more specific, they are assigned to places P_0 and P_3.

Subpartitions are also used in the case of *nested parallelism*. At subsequent nesting levels, a subpartition defines the place list for a thread to be used. The subpartition serves as the starting point for the place list at the specific level.

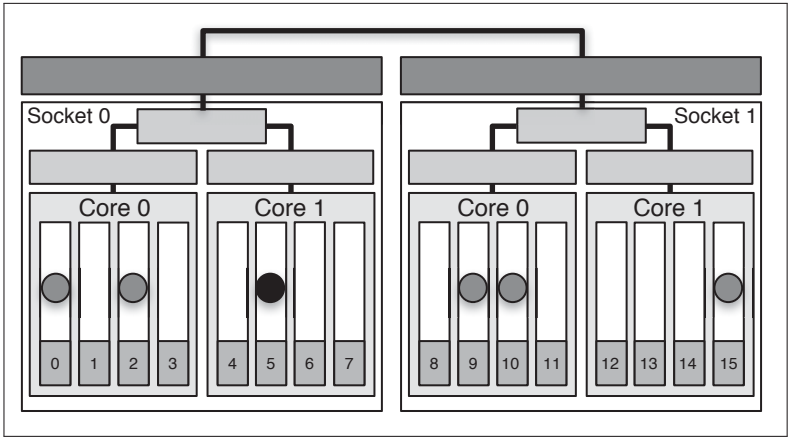

Figure 4.19: **The spread affinity policy with 6 threads and a different place list** – The place list has been modified to guarantee that core 0 on both sockets always executes two threads, while the other two cores execute a single thread. In this example it is assumed that the parent of the master thread executes in place P_2, but the number of threads per core does not depend on this. Regardless of where the master thread executes, the distribution is the same.

4.6.4 What's in a Name?

Despite their names, the `close` and `spread` affinity policies are not always as different as they may seem. Their differences depend on T, the number of OpenMP threads, and P, the number of places in the place list.

For example, in the above case with six threads and the modified place list ($P = 6$), the `close` policy may also be used to distribute the threads. The $T \leq P$ algorithm applies and each thread gets assigned to one of the six places. This results in the same mapping as shown in Figure 4.19 for the `spread` policy.

Upon closer inspection it is easy to see this is no coincidence. If the place list is the same and $T = P$, the `close` and `spread` policies result in identical mappings.

This may even be extended to the case $T \geq P$. The reason is that, if $T > P$, each of the subpartitions used by the `spread` policy, consists of a single place only. This is, however, exactly the same situation that arises if the `close` policy is applied. With both policies, the threads are distributed over the places in a consecutive way and the outcome is the same.

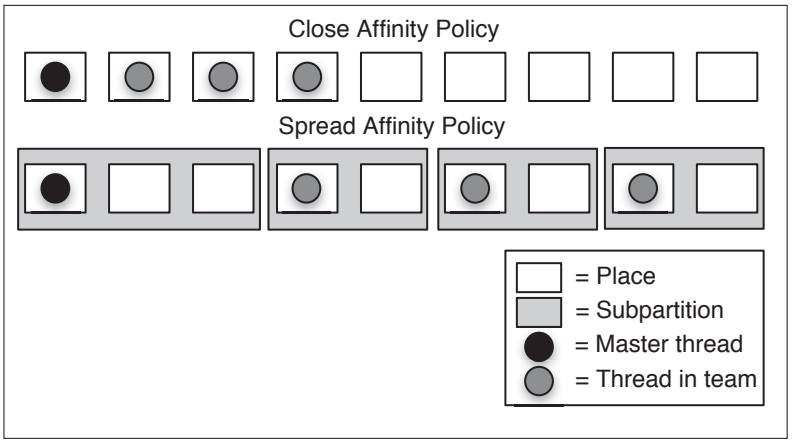

Figure 4.20: **Thread mappings in case there are four threads and nine places** – In this case, $T = 4$ and $P = 9$. Using the `close` affinity policy, the threads are mapped onto consecutive places. This is in contrast with the `spread` policy. There are four subpartitions and three of these contain two places, while one consists of three places. Threads are assigned to the subpartitions and mapped onto places as far apart as possible.

The one possible difference that may arise is if P does not divide T evenly. Then, not every subset of the threads has the same number of threads and the implementation determines how to map these subsets onto the place. This could be different for the two policies.

The policies are definitely different in the case of $T < P$. This is illustrated in Figure 4.20, when there are four threads and nine places. With the `close` policy, the four threads are scheduled onto the first four places. This is as close as possible within the context of places. In the case of the `spread` policy it is entirely different. Four subpartitions are created. One has three places, while the other three have two places. Threads are scheduled onto the subpartitions and within a subpartition the first place is selected. This maximizes the distance in terms of places, justifying the name "spread."

4.7 Making it Easier to Use the Thread Affinity Policies

The place list, places, and resource numbers may be used to provide full control over where threads execute. Now it is time to show a much easier, and more elegant way to control the placement of threads.

This ease-of-use is provided by using the *abstract names* to define the place list. These abstract names have already been touched upon in Section 4.5.3, but in this section, more details and several examples are given.

Recall there are three such names: `sockets`, `cores`, and `threads`. These names represent the various hardware execution elements in a contemporary parallel system. The interpretation of the abstract names is implementation-dependent, but it safe to assume they provide the functionality suggested by the names.

It is, however, conceivable that an architecture has additional, or different, elements not easily described by these names. This is why an implementation has the freedom to support additional abstract names, but these are system-dependent. Refer to the documentation of the specific OpenMP compiler and runtime system for a description of such extensions, if any.

The abstract names provide less control over the placement, but in often that is not needed. The considerable advantages are portability and robustness against errors in the place list. For example, setting `OMP_PLACES=cores`, is a generic choice and may be used on any system that supports the concept of cores.

This is why usage of the abstract names should be considered first, before using the system-dependent resource numbers to control the placement. These two approaches are mutually exclusive.

The abstract name is specified through the same environment variable `OMP_PLACES`, that is used to define the place list with the resource numbers. The difference is that, instead of using places defined through resource numbers, an abstract name is used. This is an example of how to use all cores in the system: `OMP_PLACES=cores`.

The abstract name is handled by the OpenMP system and transparently translated into a place list. This is why the term "place list" is used sometimes in combination with the abstract name(s).

The generated place list is not visible to the user. It is created by the system and spans all corresponding hardware scheduling elements specified. In the above example, the (hidden) place list consists of all cores in the system.

When not all elements must be used, the abstract name may be appended by a positive number in parentheses. This number is used as the length of the place list. For example, this restricts the place list to four cores: `OMP_PLACES=cores(4)`

If fewer execution units (for example cores) than available are specified, the decision of which one(s) to select is determined by the implementation. If the number exceeds the number of units available, the length of the resulting place list to be used is implementation-dependent.

This is why usage of a specific number with the abstract name may be at odds with portability. The selection of cores depends on the OpenMP runtime system, and the number of cores requested may not be available on a different system. There is nothing wrong with that, but it is good to keep in mind, in case one needs to run the same application on a system with a different configuration.

As with the explicit place list, the abstract name may be used in conjunction with the environment variable `OMP_PROC_BIND`, or the `proc_bind` clause on the parallel region. Through these controls, the affinity policy (`master`, `close`, or `spread`) is defined. In the next three subsections, it is shown how to use the abstract names in combination with the affinity policies.

As before, the generic, multi-core system described in Figure 4.1 on page 152 is the target and the notation introduced in Section 4.6 is used.

To simplify the examples and diagrams, the assumption is that the parent of the master thread *always* executes in the first place of the (hidden) place list.[11]

4.7.1 The Sockets Abstract Place List

The `sockets` abstract name is used to schedule OpenMP threads across the sockets in the system. It is specified as follows: `OMP_PLACES=sockets`. The generated place list consists of all the sockets in the system. A socket may have multiple cores and more than one hardware thread per core.

The next step is to decide upon the affinity policy to control where the OpenMP threads are scheduled to run. This is illustrated with an example using the generic, multi-core system. It has two sockets and, in this case, the place list consists of two places ($P = 2$). Assume there are six OpenMP threads ($T = 6$).

The `master` policy is very straightforward. All six threads are scheduled to run on the same socket where the master thread executes. Because in this example the

[11] In the remainder of this section, it is no longer explicitly mentioned that this list is generated by the system and hidden to the user.

parent of the master thread is assumed to run on socket 0, all threads execute on this socket. It is up to the scheduler to decide which of the available eight hardware threads to select for the six OpenMP threads to run on.

In the case of the `close` policy, the "$T > P$" algorithm, described in Section 4.6.2, applies. Following the algorithm for this case, there are two thread subsets ST_0 and ST_1. They contain the following OpenMP thread numbers:

$$ST_0 = \{0, 1, 2\}$$
$$ST_1 = \{3, 4, 5\}$$

The threads in ST_0 execute in socket 0, while the threads in ST_1 are scheduled to run on socket 1. As mentioned earlier, it is up to the scheduler to select the cores and hardware threads. This means various mappings are allowed.

This is illustrated in Figure 4.21. In the first mapping, the threads within a socket execute in the same core. This is not the case in the second mapping, where the OpenMP threads are scattered over all cores. Again, both are valid mappings and also not the only ones possible.

As shown in Section 4.6.4, the mapping using the `spread` affinity policy is the same, as long as $T \geq P$, which is the case here. In the case of $T < P$, these policies lead to a different mapping, but in this particular scenario that is a corner case. There are only $P = 2$ places and only if $T = 1$, does this condition apply. If so, the mapping is again the same.

Whether or not the variation in the mapping of threads onto the hardware resources is acceptable, is decided by the user. What if a finer-grained level of control is desired? This is covered in the next two sections.

4.7.2 The Cores Abstract Place List

The `cores` abstract name indicates to the system that a core should be used as the scheduling entity for the OpenMP threads. This choice is specified in the following way: `OMP_PLACES=cores`.

The generated place list consists of all the cores in the system. Although a core may have more than one hardware thread, this is not considered for the mapping policy. It can be taken into account by the runtime scheduler though.

The `cores` abstract name is used on the generic, multi-core system. The difference compared to the previous case with the `sockets` name, is that there are now

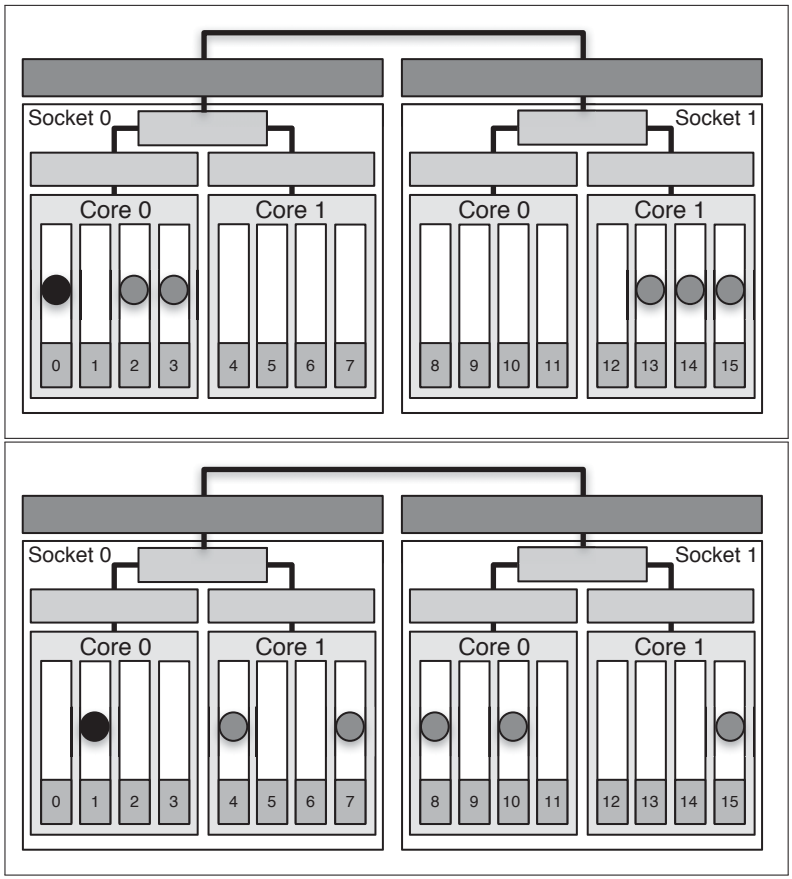

Figure 4.21: **Two possible mappings using the sockets abstract name with the close policy** – In the first mapping, the three threads per socket execute within the same core. This is, however, not guaranteed. The second mapping is equally valid, but the threads are scattered over all four cores.

four places ($P = 4$). They span across the four cores in the system. Assume six OpenMP threads ($T = 6$) again.

If the `master` affinity policy is applied, all six threads are scheduled to run on a single core. This is the same core where the parent of the master thread runs.

With the `close` policy, the "$T > P$" algorithm, described in Section 4.6.2, applies. There are $P = 4$ subsets, ST_0 through ST_3, with consecutive thread numbers, but because $floor(T/P) = 1$ and $ceiling(T/P) = 2$, not all subsets contain the same number of threads. To be more precise, two subsets contain two threads and the other two subsets contain a single thread only. The details of this distribution are implementation-dependent. For this example, assume the following:

$$ST_0 = \{0, 1\}$$
$$ST_1 = \{2, 3\}$$
$$ST_2 = \{4\}$$
$$ST_3 = \{5\}$$

These four subsets are assigned to the four places. Per the definition of the `close` policy, the threads in ST_0 are scheduled in the place where the parent of the master thread is running, which is a core in this case. The threads in ST_1 are scheduled in the next place in the place list, and so forth.

There are no requirements regarding the order of the cores in the (generated) place list, leaving a degree of freedom as to how to map the thread subsets onto the cores. In the generic, multi-core system for example, one socket may end up executing four threads, while two threads are running on the other socket.

This is actually what may be desired using the `close` policy, but an equally valid mapping is to execute three threads per socket. Both scenarios are illustrated in Figure 4.22.

The thing to note is that there is overlap between the mappings. The mapping shown in the second diagram in Figure 4.21 is the same as the second diagram in Figure 4.22. The difference in the hardware threads used illustrates the freedom the runtime scheduler has in assigning OpenMP threads to hardware resources. There is a difference, however. The mapping shown in the first diagram in Figure 4.21 is not possible here. Each core *must* execute at least one thread.

As explained in Section 4.6.4, in this case ($T \geq P$), the `spread` affinity policy has the same mapping characteristics. A difference arises if $T < P$. Assume the place list contains the cores in the left to right order, as shown in the generic, multi-core architecture. If two threads are used ($T = 2$), and the `close` policy is applied, each core on one of the two sockets executes a thread. If the `spread` policy is used, each socket executes one thread. Although not guaranteed by the OpenMP

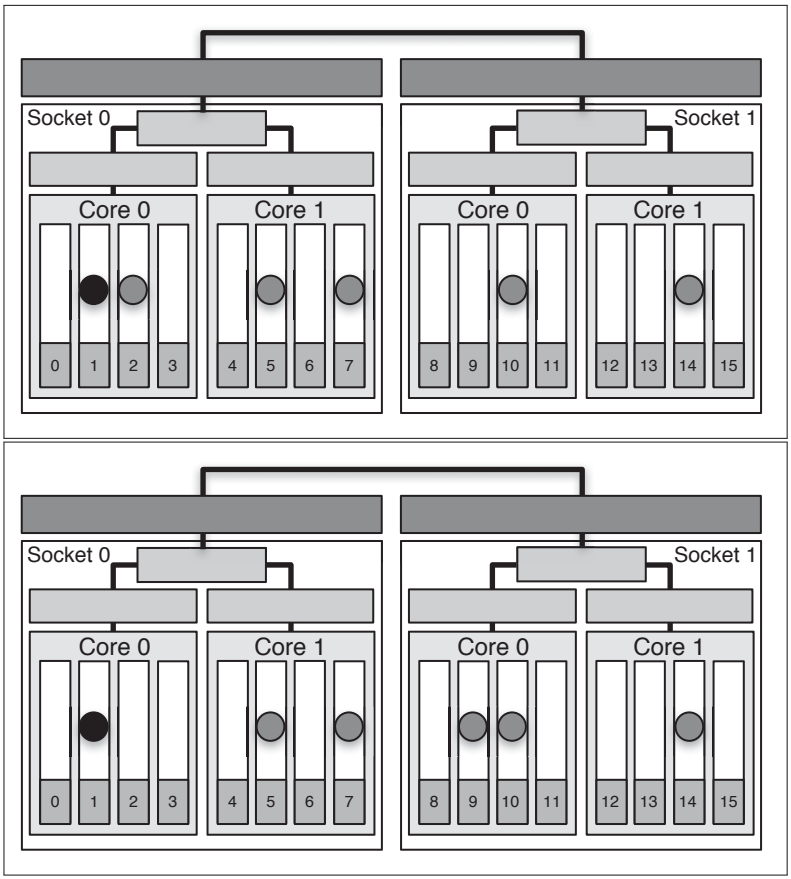

Figure 4.22: **Two possible mappings using the cores abstract name with the close policy** – In the first mapping, the first socket executes four threads, while there are only two threads running in the second socket. The second mapping is equally valid, but the threads are more balanced over the system.

4.5 specifications, one may assume that implementations generate topology-aware place lists. This is at least what is assumed to be the case here.

Although we have narrowed down where the OpenMP threads are allowed to run, there is still an uncertainty. The next section shows how the `threads` abstract name provides full control over the mapping.

4.7.3 The Threads Abstract Place List

The `threads` abstract name expands to a place list that contains all the hardware threads in the system. This name is selected as follows: `OMP_PLACES=threads`.

Most likely, the `master` policy is not very meaningful here, because it implies all OpenMP threads in the team are executing on a single hardware thread.

In general, using more OpenMP threads than hardware threads, is likely to degrade performance. This means that with the `threads` abstract name, the practical scenarios are for the "$T \leq P$" case only. These are the ones discussed here.

With the `close` policy, there are sufficient places for the OpenMP threads to run. Using the notation from Figure 4.10 on page 170, it is easy to see that $T_i \in P_i$ for all threads $i = 0, \ldots, T-1$. In terms of the places in the place list, the OpenMP threads execute back-to-back and are indeed "close" in this sense.

As before, the `spread` policy has T subpartitions in the case of $T \leq P$. With the choice of `threads` for the abstract name, the subpartitions consist of one or more hardware threads. Assignment of OpenMP threads to places is at this granularity. The lower the number of threads, the more they are spaced apart.

As observed in Section 4.6.4, the `close` and `spread` policies are equal if $T \geq P$.

Consider an example with six threads ($T = 6$) on the two-socket, generic, multicore system. There are 16 hardware threads ($P = 16$), so consequently, $T < P$. Assume the places in the generated place list follow the left to right numbering of the hardware threads in the system.

If the `close` policy is applied, OpenMP threads are assigned to consecutive hardware threads, starting with the hardware thread in which the parent of the master thread runs. This is shown in Figure 4.23, where the parent of the master thread of the team executes on hardware thread 0.

With the `spread` policy there are six subpartitions ($T = 6$). Four of these contain $ceiling(16/6) = 3$ places, while the remaining two subpartitions contain $floor(16/6) = 2$ places. Each OpenMP thread is assigned to a subpartition. A possible mapping is the following:

$$SP_0 = \{P_0, P_1, P_2\} = \{0, 1, 2\} \text{ and } T_0 \in SP_0$$

$$SP_1 = \{P_3, P_4, P_5\} = \{3, 4, 5\} \text{ and } T_1 \in SP_1$$

$$SP_2 = \{P_6, P_7, P_8\} = \{6, 7, 8\} \text{ and } T_2 \in SP_2$$

$$SP_3 = \{P_9, P_{10}, P_{11}\} = \{9, 10, 11\} \text{ and } T_3 \in SP_3$$

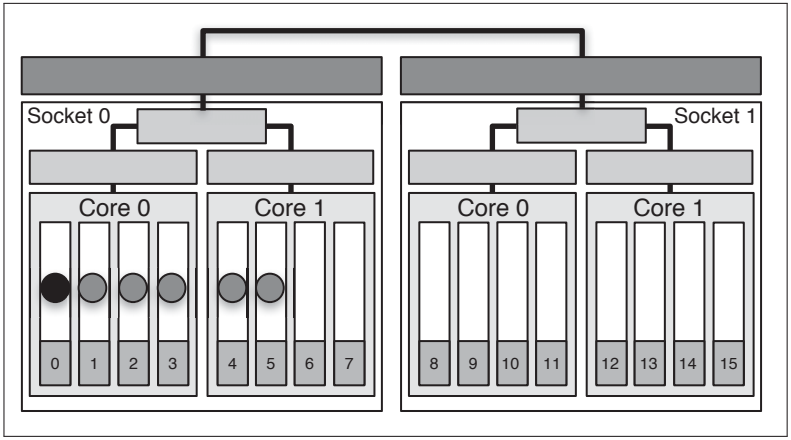

Figure 4.23: **Mapping using the threads abstract name with the close policy** – If the parent of the master thread executes on hardware thread 0, the six OpenMP threads execute on hardware threads $0-5$.

$$SP_4 = \{P_{12}, P_{13}\} = \{12, 13\} \text{ and } T_4 \in SP_4$$

$$SP_6 = \{P_{14}, P_{15}\} = \{14, 15\} \text{ and } T_5 \in SP_5$$

The reason the word "possible" is used here is, because the implementation determines which subpartitions have three places and which one has two places. See also Section 4.6.3, starting on page 178.

Figure 4.24 illustrates two mappings for the subpartitions given above. As mentioned several times before, the assignment of OpenMP threads within a place is implementation-dependent.

In the first case, the distribution of the threads over the system is optimal. The second example, although valid as well, is less favorable. One core is not used at all and socket 1 executes twice as many OpenMP threads as socket 0.

Although the number of undesirable mappings has been reduced, there are still less optimal scenarios. If more control over the placement is required, the solution is to define an explicit place list. Figure 4.19, on page 186 shows such an example.

For a different number of threads, things may work out very well. For example, with eight threads ($T = 8$), each of the eight subpartitions contains $16/8 = 2$ consecutive hardware threads. The OpenMP threads are mapped symmetrically

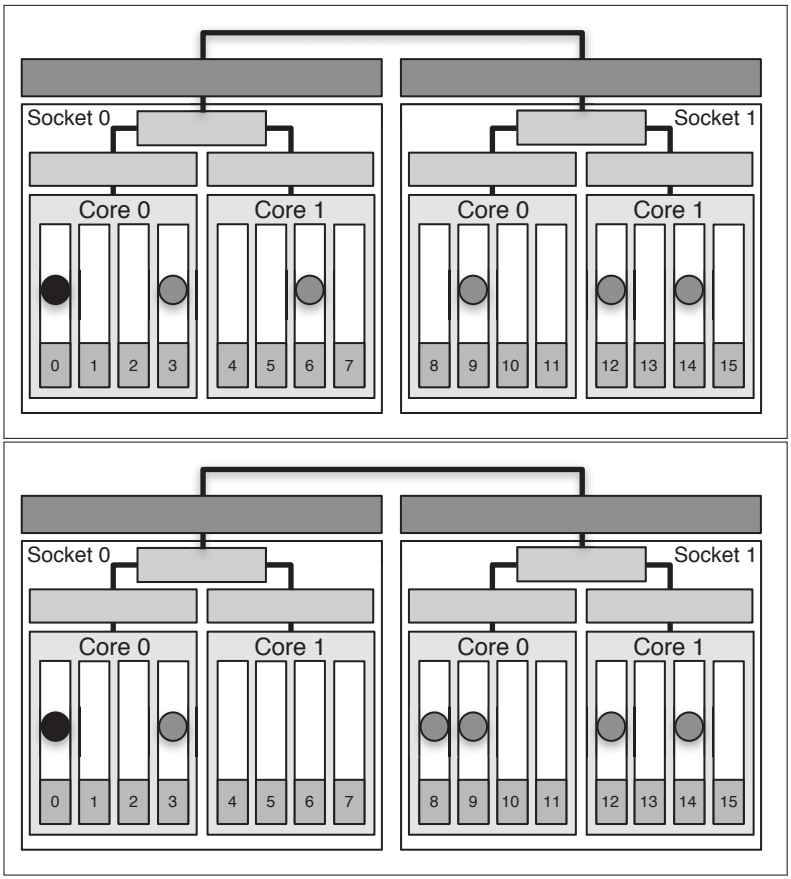

Figure 4.24: **Two possible mappings using the threads abstract name with the spread policy** – The distribution shown in the first picture is optimal. Each core executes one, or two, threads and the threads are well-balanced over the system. The second mapping, although possible, is not optimal. One core is not used at all and the number of threads per socket is not balanced.

onto these subpartitions, effectively spreading them out equally. An example of this is shown in Figure 4.25.

This is not a coincidence. In general, placement is more precisely defined if the number of threads T and the number of places P divide equally. If this is not

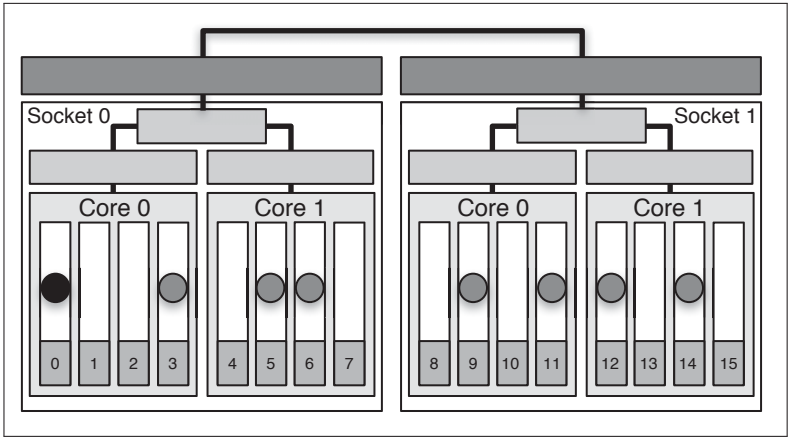

Figure 4.25: **The threads abstract name with the spread policy and eight threads** – In this case, the OpenMP threads are spread equally over the hardware threads in the system, maximizing the bandwidth.

the case, a load balancing issue arises and the implementation needs to make more decisions where to place the threads.

4.8 Where Are My Threads Running?

So far, the examples, have been theoretical. The time has come to show how thread affinity works in practice, but first some background information needs to be provided.

As of the OpenMP 4.5 specifications, diagnostic runtime functions to query the thread affinity environment and placement information, are supported. These are introduced and discussed in Section 2.3.2, starting on page 64.

These functions may be used to instrument the application and print the affinity information to verify where a thread executes. The drawback is that these calls must be included in the source code.

For the purpose of demonstrating how affinity works in practice, we prefer to do this without instrumenting the code.[12] This is why for this section, we turned to

[12]Most likely, the next release of the specifications supports an environment variable to print diagnostic thread affinity information at runtime.

Reference	Place List	Places	Coverage
PL_1	{0:128:1}	1	All hardware threads in socket 0.
PL_2	{0:8:1}:16:8	16	All hardware threads in socket 0.
PL_3	{0:8:1}:32:8	32	All hardware threads in the system.

Figure 4.26: **Definitions and references for the place lists used** – Throughout this section, several place lists are used. They are defined in this table and a reference for easy identification is assigned to each of them. The third column lists the number of places in the place list. The last column describes which of the hardware threads are included in the place list.

DTrace, a tool that allows the user to obtain runtime diagnostics at the OS-level, without the need to explicitly instrument the application. It is available on the Oracle Solaris, Oracle Linux, and macOS operating systems.[13] More information on DTrace can be found in the Glossary at page 340.

A DTrace script to monitor the mapping of the OpenMP threads onto the hardware threads was developed. The results are shown in the examples below, but the details of this script are beyond the scope of this book. Two tables are printed. The interpretation of these is given in Section 4.8.1 on page 200.

The system used in all the thread placement monitoring experiments has two sockets. Each socket has 16 cores, with eight hardware threads per core. This means it has a total of 32 cores and 256 hardware threads. The output listed in Figure 4.6 on page 163, was obtained using this system.

In this section, several place lists are used. For ease of reference they have been given a name and are listed in Figure 4.26. Note the difference between PL_1 and PL_2. Although both span all the hardware threads in socket 0, the first list has a single place only, while the second list consists of 16 places.

4.8.1 Affinity Examples for a Single-Level Parallel Region

The DTrace script has been used to obtain affinity statistics for a single-level parallel region. Within this parallel region there is only one `single` region with a print statement. The relevant code fragment is shown in Figure 4.27. This simple test program has been used to verify several affinity scenarios.

[13] Linux has a similar tool called *SystemTap* [25], but this has not been explored further.

```
1 #pragma omp parallel
2 {
3    #pragma omp single
4    {
5       printf("Single region executed by thread %d\n",
6              omp_get_thread_num());
7    } // End of single region
8 } // End of parallel region
```

Figure 4.27: **Single-level parallel region used to demonstrate the thread affinity settings** – There is one single-level parallel region. Only one thread prints a message.

We start with an explicit OpenMP place list. The numbering scheme depends on the system, as well as the OS, and suitable commands must be used to obtain this information. On this system, hardware threads $0-127$ are on socket 0, while threads $128-255$ are on socket 1.

The interval notation is used to define the list: `OMP_PLACES="{0:128:1}"`.[14] This is place list `PL_1`, as defined in Figure 4.26. It consists of a single place only and spans the first half of the 256 hardware threads of the target system. These are all the threads in socket 0.

The next choice to be made is the thread affinity policy. In this first example, the `close` policy is selected by setting environment variable `OMP_PROC_BIND` as follows: `OMP_PROC_BIND=close`.

Four OpenMP threads are used ($T = 4$). Because there is a single place only ($P = 1$), the algorithm for the "$T > P$" case, as given in Section 4.6.2, applies. The thread subset ST_0 includes all four OpenMP threads ($T/P = 4$), and they are assigned to the single place spanning all hardware threads in socket 0.

Following this algorithm, and assuming the parent of the master thread executes on hardware thread 0, one may expect the other three threads to execute on hardware threads $1-3$, respectively. There is, however, *no guarantee* this happens. The order within a place list does not matter, and the implementation is free to select any hardware thread in the place list.

In Figure 4.28, the runtime statistics from the DTrace script are shown. Here, and in all subsequent figures in this section, two tables are listed. Together, they show

[14] How to set the environment variable, depends on the OS and shell used.

```
Thread   On HW Thread   Node   Created Thread
     0              0      1                1
     0              0      1                2
     0              0      1                3

Thread  Running on HW Thread    Bound to HW Thread
     0                   178                     0
     1                     0                     1
     2                     0                     2
     3                     0                     3
```

Figure 4.28: **Affinity statistics for the PL_1 place list and the close policy** – The first table shows that thread 0 creates the other three threads. It was running on hardware thread 0 when it did so. The second table shows it did not start there, but was migrated from hardware thread 178. This table also shows that the other threads started on the same hardware thread 0, but were then bound to hardware threads $1 - 3$.

what has happened both at the OpenMP thread level, as well as which hardware threads are involved.

The first table shows thread creation statistics and has four columns. The first column gives the OpenMP thread ID. The second column, called "On HW Thread," lists the ID of the hardware thread this OpenMP thread was running on when it created other threads. Column "Node" gives the memory node reference ID.

In Linux, this is called a "NUMA Node," while Solaris calls it a "Locality Group," or "lgroup" for short. In both cases, such a node has a uniform memory access time for the hardware thread(s) connected to it. It is considered to be local to those threads. Column four, "Created Thread," lists the OpenMP thread that was created by the thread in the first column.

The second table indicates whether an OpenMP thread moved ("migrated") from one hardware thread ("Running on HW Thread") to another ("Bound to HW Thread"). Interpretation of these results sometimes requires cross checking the two tables. One thing missing from these tables is a timestamp. The full script includes this information, and in some examples this is used to explain the sequence of events.

The first table shows that thread 0 created the other three threads. At the time of thread creation, this thread was executing on hardware thread 0. Apparently, this hardware thread is connected to memory node 1.

```
Thread   On HW Thread   Node   Created Thread
     0              0      1                1
     0              0      1                2
     0              0      1                3

Thread  Running on HW Thread    Bound to HW Thread
     0                   244                     0
     1                     0                     8
     2                     0                    16
     3                     0                    24
```

Figure 4.29: **Affinity statistics for the PL_2 place list and the close policy** – The results demonstrate this policy and how it relates to the place list. With four threads, the first four places are used to schedule the OpenMP threads. In this case, each thread is placed onto the first hardware thread within a place, but as explained earlier, which hardware thread to select is an implementation-dependent choice. Effectively, each thread is executing on a different core.

The second table shows that thread 0 started on hardware thread 178 but migrated to hardware thread 0. This is interesting, because this demonstrates that the master thread started on the second socket, and was moved to socket 0. The other three OpenMP threads started on hardware thread 0, but were moved, or migrated, to hardware threads $1-3$.

These results show the implementation does what the `close` policy is meant to achieve. Given the definition of the place list, any hardware thread ID in the range $0-127$ is actually valid.

In the second example, place list `PL_2` is used. There are 16 places and each spans the hardware threads within a different core in socket 0. This results in the following expansion for the place list: $P_0 = \{0, 1, \ldots, 7\}$, $P_1 = \{8, 9, \ldots, 15\}, \ldots,$ $P_{15} = \{120, 121, \ldots, 127\}$.

The same example has been executed using this second place list and the `close` policy. Because there are four OpenMP threads ($T = 4$) and 16 places ($P = 16$), the algorithm for the "$T \leq P$" case for the `close` policy applies. This implies that the OpenMP threads are assigned to consecutive places, starting with the place where the parent thread of the master thread runs. Each place spans a core, and the threads must be assigned to the individual cores in socket 0. The affinity statistics are shown in Figure 4.29.

As before, OpenMP thread 0 creates the other threads. It apparently started at hardware thread 244. From there, it migrated to thread 0. Threads $1-3$ started on hardware thread 0, but migrated to hardware threads 8, 16 and 24 respectively. This shows that all OpenMP threads are indeed scheduled onto successive places. Per the definition of the places, each thread executes on its own core on socket 0, and the implementation selected the first hardware thread on each core.

There is an important question to address: *"Why was the first place from the place listed selected, and not the place containing hardware thread* 244*?"*

The reason is that this hardware thread is not part of the place list. It is a thread on core 30, which is on socket 1. Upon program start-up, the OS has no notion of the OpenMP affinity settings and apparently decided to start the application on hardware thread 244. The OpenMP runtime system is, however, well aware of the affinity settings and moved the thread to hardware thread 0 *before* executing the parallel region.

The specifications do not guarantee this, though. Not all systems support thread migration, and the action taken in the above scenario is system-dependent. In general, if an affinity request cannot be fulfilled, the affinity of threads in the team is implementation-defined.

On this system, thread migration is supported, however, and the runtime system may move the master thread to the right location, which in this case is place P_0 of the place list. This marks the starting point of the places to be assigned to subsequent threads.

The same code and place list were used in the next experiment. The difference is that the `spread` affinity policy is selected. Following the algorithm for the "$T \leq P$" case, it follows there are $T = 4$ subpartitions with $P/T = 4$ consecutive places each:

$$SP_0 = \{P_0, P_1, P_2, P_3\}$$
$$SP_1 = \{P_4, P_5, P_6, P_7\}$$
$$SP_2 = \{P_8, P_9, P_{10}, P_{11}\}$$
$$SP_3 = \{P_{12}, P_{13}, P_{14}, P_{15}\}$$

Expanding the individual places to their lists of hardware threads, results in the table shown in Figure 4.30.

The mapping algorithm guarantees that the four OpenMP threads are assigned to the first place in each of the subpartitions selected for that thread. For example, because of the assumption of where the parent of the master thread executes, thread

Subpartition	Places			
SP_0	$\{0,\ldots,7\}$	$\{8,\ldots,15\}$	$\{16,\ldots,23\}$	$\{24,\ldots,31\}$
SP_1	$\{32,\ldots,39\}$	$\{40,\ldots,47\}$	$\{48,\ldots,55\}$	$\{56,\ldots,63\}$
SP_2	$\{64,\ldots,71\}$	$\{72,\ldots,79\}$	$\{80,\ldots,87\}$	$\{88,\ldots,95\}$
SP_3	$\{96,\ldots,103\}$	$\{104,\ldots,111\}$	$\{112,\ldots,119\}$	$\{120,\ldots,127\}$

Figure 4.30: **The expanded subpartition definitions** – This table shows the four subpartitions and their place list with hardware thread numbers. This is for the scenario with $T = 4$ threads, a place list defined by `OMP_PLACES="{0:8:1}:16:8"`, and the `spread` affinity policy.

```
Thread   On HW Thread   Node   Created Thread
     0              0      1                1
     0              0      1                2
     0              0      1                3

Thread  Running on HW Thread    Bound to HW Thread
     0                   208                     0
     1                     0                    32
     2                     0                    64
     3                     0                    96
```

Figure 4.31: **Affinity statistics for the PL_2 place list and the spread policy** – As before, the application is migrated to hardware thread 0. From there, the other threads are created and migrated to their destination hardware thread. Each thread is indeed scheduled onto the first place in its subpartition.

2 is assigned to SP_2 and it is scheduled onto a hardware thread in the range $64-71$ within this subpartition.

The runtime affinity results for this setup are shown in Figure 4.31 and clearly demonstrate how this policy works. As seen before, the application starts at a different hardware thread than the target. In this case, it starts at hardware thread 208. From there it is migrated to hardware thread 0 and creates the other three threads on the same hardware thread. From there, these threads migrate to their destination hardware threads.

Thread	Running on HW Thread	Bound to HW Thread
0	111	0
1	0	16
2	0	32
3	0	48
4	0	64
5	0	80
6	0	96
7	0	112

Figure 4.32: **Affinity statistics for the PL_2 place list, the spread policy, and 8 threads** – In this case, all eight subpartitions contain two consecutive places. Each OpenMP thread is mapped onto the first hardware thread in the first place of the corresponding subpartition.

As expected, those three threads are scheduled onto subpartitions SP_1, SP_2 and SP_3 respectively. Comparing the destination hardware threads with those listed in Figure 4.30 it is shown that they are scheduled onto the first hardware thread from the first place in the respective subpartition.

To further demonstrate this policy, the experiment was repeated, but with the number of threads set to eight ($T = 8$). This results in 8 subpartitions, containing $P/T = 2$ consecutive places each. For example, subpartitions $SP_0 = \{P_0, P_1\}$ and $SP_1 = \{P_2, P_3\}$. As before, the first OpenMP thread is assigned to SP_0, the second thread to SP_1, and so on. The affinity results of this experiment are given in Figure 4.32.

In the next two figures the first table with the thread creation statistics is omitted. It is similar to what we have seen in the previous experiments.

The results confirm the settings work as expected. Each OpenMP thread is mapped onto the first place of its subpartition. For example, thread 4 is assigned to subpartition $SP_4 = \{P_8, P_9\}$, and it executes on hardware thread 64, which is the first hardware thread in P_8.

Until now, the place list was restricted to span a single socket only. To demonstrate how to use both sockets, the following place list is defined: `OMP_PLACES="{0:8:1}:32:8"`. This place list consists of 32 places with 8 hardware threads per place: $P_i = \{i * 8, i * 8 + 1, \ldots, i * 8 + 7\}$ for $i = 0, \ldots, 31$. It is place list `PL_3` from Figure 4.26.

```
Thread  Running on HW Thread    Bound to HW Thread
     0                   162                     0
     1                     0                    32
     2                     0                    64
     3                     0                    96
     4                     0                   128
     5                     0                   160
     6                     0                   192
     7                     0                   224
```

Figure 4.33: **Affinity statistics for the PL_3 place list, the spread policy and 8 threads** – There are again eight subpartitions, but this time with four consecutive places each. The OpenMP threads are mapped onto the first hardware thread in the first place of the corresponding subpartition.

With the `spread` policy, the algorithm for the "$T \leq P$" case applies. This results in eight subpartitions $SP_0, \ldots, SP_7$. Each subpartition contains $P/T = 4$ consecutive places. It is easy to see that the first place in subpartition SP_j is P_{j*4} for $j = 0, \ldots, 7$. It follows from the algorithm that OpenMP thread j is assigned to this place.

The observed runtime results for the `spread` affinity policy and eight threads are listed in Figure 4.33. The first hardware thread within place P_{j*4} has number $j*4*8 = j*32$ ($j = 0, \ldots, 7$). The results show that each OpenMP thread is scheduled onto this hardware thread. As mentioned before, this is an implementation-dependent choice. Every hardware thread selected from the correct place is equally valid.

4.8.2 Affinity Examples for a Nested Parallel Region

In this section, several examples for a two-level nested parallel region are presented and discussed. The relevant code fragment used in the experiments is listed in Figure 4.34.

The outer-level parallel region spans lines $1 - 16$. Only one thread prints a message. The second-level parallel region starts at line 8 and ends at line 15. Within this region, a single thread prints a message.

```
 1 #pragma omp parallel
 2 {
 3    #pragma omp single
 4    {
 5      printf("First single region executed by thread %d\n",
 6             omp_get_thread_num());
 7    } // End of single region level 1
 8    #pragma omp parallel
 9    {
10       #pragma omp single
11       {
12         printf("Second single region executed by thread %d\n",
13                omp_get_thread_num());
14       } // End of single region level 2
15    } // End of parallel region level 2
16 } // End of parallel region level 1
```

Figure 4.34: **Nested parallel region used to demonstrate the thread affinity settings** – The nesting level is two. Regardless of the number of threads used, only two messages are printed, one from each nesting level.

In the first scenario, place list `PL_1` (see also Figure 4.26 on page 198) is used. This list consists of a single place that spans all the hardware threads on socket 0 of the two-socket system.

This is a nested parallel region and, unless it is acceptable to rely on the implementation-dependent defaults, the affinity policy on both levels must be specified. In this case, we set `OMP_PROC_BIND="spread,close"`. For the outer parallel region, two threads are used. The second-level parallel region has four threads. This is set as follows: `OMP_NUM_THREADS="2,4"`.

The place list contains one place only ($P = 1$). At the outer level, there are two threads and the algorithm for the "$T > P$" scenario of the `spread` affinity policy applies. This implies there is only a single subpartition (the entire place) and all threads are assigned to it. In other words, two hardware threads on socket 0 are selected at runtime.

At the second level, the `close` policy applies. Each of the two threads creates its own team of threads and have the role of the master thread in their respective teams. The place list has one place only and in both cases, the same original

```
Thread   On HW Thread   Node   Created Thread
     0              0      1                1
     0              0      1                3
     0              0      1                5
     0              0      1                7
     1              1      1                2
     1              1      1                4
     1              1      1                6

Thread  Running on HW Thread    Bound to HW Thread
     0                   191                     0
     1                     0                     1
     2                     1                     6
     3                     0                     7
     4                     1                     2
     5                     0                     3
     6                     1                     5
     7                     0                     4
```

Figure 4.35: **Affinity statistics for a nested parallel region using place list PL_1** – There are $2x4$ threads and the `"spread,close"` policy is applied. The two outer-level threads are mapped onto hardware threads 0 and 1. The other six threads at the second level are executed on hardware threads $2-7$.

place list is used. There are four threads per team and again the algorithm for the "$T > P$" scenario applies. Because there is one place only, the additional three threads per team are mapped onto the same place list.

The results in Figure 4.35 show that the program started execution on hardware thread 191 and from there migrated to hardware thread 0. OpenMP thread 1 was created on hardware thread 0 and moved to hardware thread 1. Both are part of the same node.

These two threads created their own team with three additional threads each. The first team consists of OpenMP threads $0, 3, 5$, and 7. Threads $1, 2, 4$, and 6 constitute the second team. Although these new threads were created on the hardware thread that their master thread was running on, they were moved to their destination. In the end, the eight threads are executing on hardware threads $0-7$.

It is interesting to note that this is a degenerate case. The `spread` and `close` policies are not really doing anything different. Actually, setting `OMP_PROC_BIND="master,master"` results in the same mapping of the OpenMP threads onto the hardware threads.

In the second example, the mapping is very different. The program, the number of threads per level, and the affinity policies are the same, but the place list is set to `PL_2`. There are 16 places ($P = 16$) now and each place spans a core in socket 0. For both policies, the "$T \leq P$" scenario applies.

At the first level there are two subpartitions and they contain $16/2 = 8$ consecutive places:

$$SP_0 = \{P_0, \ldots, P_7\} = \{\{\texttt{0:8:1}\}\texttt{:8:8}\}$$
$$SP_1 = \{P_8, \ldots, P_{15}\} = \{\{\texttt{64:8:1}\}\texttt{:8:8}\}$$

The `spread` policy is applied to the outer level. Following the algorithm for this case, it is easy to see that OpenMP thread 0 executes in place P_0. Thread 1 is scheduled to run in place P_8. Given the definitions of these places, this means that thread 0 runs on core 0 and thread 1 executes on core 8.

At the second level, the two place lists are SP_0 and SP_1. Each has eight places ($P = 8$) and there are four threads per team ($T = 4$). Again, the "$T \leq P$" scenario applies, but now for the `close` policy. As a result, successive threads in a team are scheduled onto consecutive places in their respective place list.

For example, the four threads in the second team are mapped onto places P_8, $\ldots$, P_{11}, and they execute on cores $8, \ldots, 11$. As always, the choice of a hardware thread within a place is system-dependent.

The results of the experiment are shown in Figure 4.36. The mappings show that the settings work as expected. After the initial migration from hardware thread 143 to the destination hardware thread 0, OpenMP thread 0 creates thread 1. Both threads execute in the same socket, but far apart.

Although not explicitly clear from this overview, thread 0 then creates threads 3, 5, and 7 at the start of the second-level parallel region. Thread 1 creates threads 2, 4, and 6. The threads in the first team are scheduled onto hardware threads $i * 8$ for $i = 0, \ldots, 3$. The threads in the second team execute on hardware threads $64 + i * 8$ for $i = 0, \ldots, 3$.

The place list $P_i = \{i*8, i*8+1, \ldots, i*8+7\}$ for $i = 0, \ldots, 7$. It follows that the hardware thread numbers used correspond to the first hardware thread in places $P_0, \ldots, P_3$ for the first team and $P_8, \ldots, P_{11}$ for the second team.

```
Thread   On HW Thread   Node   Created Thread
     0              0      1                1
     0              0      1                3
     0              0      1                5
     0              0      1                7
     1             64      1                2
     1             64      1                4
     1             64      1                6

Thread  Running on HW Thread    Bound to HW Thread
     0                   143                     0
     1                     0                    64
     2                    64                    72
     3                     0                     8
     4                    64                    80
     5                     0                    16
     6                    64                    88
     7                     0                    24
```

Figure 4.36: **Affinity statistics for a nested parallel region using place list PL_2** – There are $2x4$ threads and the `"spread,close"` policy is applied. The two outer-level threads are spread out as much as possible, and mapped onto hardware threads 0 and 64. The threads at the second-level are executed as close as possible to their respective master thread.

The third and last example is based upon the previous case, but this time, place list `PL_3` is used. This means there are 32 places, and each place spans the hardware threads in a core.

Following the same algorithms, there are again two subpartitions at the first level ($T = 2$) and 16 consecutive places in each subpartition. This time, a subpartition spans all the cores and hardware threads in a socket. As a result, each outer-level thread executes in its own socket. The threads in the two teams, created at the second level, are kept close. Per the definitions of the place list and subpartitions, this means that each OpenMP thread executes on a separate core.

The runtime affinity statistics in Figure 4.37 confirm the above. There are $2x4$ threads and the `"spread,close"` policy is specified. The two outer-level OpenMP

```
Thread   On HW Thread   Node   Created Thread
     0              0      1                1
     0              0      1                2
     0              0      1                4
     0              0      1                6
     1            128      2                3
     1            128      2                5
     1            128      2                7

Thread  Running on HW Thread    Bound to HW Thread
     0                   255                     0
     1                     0                   128
     2                     0                     8
     3                   128                   136
     4                     0                    16
     5                   128                   144
     6                     0                    24
     7                   128                   152
```

Figure 4.37: **Affinity statistics for a nested parallel region using place list PL_3** – There are $2x4$ threads and the `"spread,close"` policy is specified. The two outer level threads are spread out as much as possible and the threads at the second level are executed as close as possible to their respective master thread.

threads execute on hardware threads 0 and 128. These are in a different socket.[15] The two thread teams at the second level execute on four consecutive cores on each socket.

4.8.3 Making Things Easier Again

The previous two examples demonstrated how the thread placement in a nested parallel region may be controlled. In this section, it is demonstrated how the abstract names may be used to define the mappings. The example code, and all settings except for the place list definition, are the same. The place list definitions for all three cases discussed next are given in Figure 4.38.

[15] This is consistent with the difference in the node number.

Reference	Place List	Places	Coverage
Case I	`sockets(1)`	1	One socket in the system.
Case II	`sockets(2)`	2	Two sockets in the system.
Case III	`cores`	32	All cores in the system.

Figure 4.38: **Place list definitions and properties** – The second column contains the setting for environment variable `OMP_PLACES`.

Thread mappings for Case I In the first case, there is only a single place, spanning all hardware threads within a socket. The runtime affinity statistics shown in Figure 4.39, are similar to those from Figure 4.35, because in both cases the place list spans all hardware threads in a single socket.

The only possible difference is the selection of the socket. With place list `PL_1`, this was hardcoded to socket 0, while the OpenMP runtime system selects a socket. The output shows that the master thread at the first level was initially running on hardware thread 43 and was moved to its destination, hardware thread 0. Both hardware threads are on socket 0 of this system. Apparently the runtime system selected the first socket on which to run. This is the same starting point as the example shown in Figure 4.35, and the mapping is the same as in this earlier case.

Thread mappings for Case II The place list in the second case specifies two sockets to be used.[16] This definition expands to a place list consisting of two places ($P = 2$). Each place spans the hardware threads within one socket.

At the outer-level, there are two subpartitions ($T = 2$), containing one place each ($P/T = 1$). As each place spans a socket, the two outer-level threads are scheduled onto a different socket. Which socket they run on depends on where the parent of the master thread is running.

Upon encountering the second-level parallel region, each of these two threads creates a team of four threads. The place list is inherited from the (new) master thread and contains a single place. In this case, the place spans the socket where the respective master thread is running.

[16] On the target system this is equivalent to simply using "sockets," because this system has two sockets only.

```
Thread   On HW Thread   Node   Created Thread
     0              0      1                1
     0              0      1                2
     0              0      1                4
     0              0      1                6
     1              1      1                3
     1              1      1                5
     1              1      1                7

Thread  Running on HW Thread    Bound to HW Thread
     0                    43                     0
     1                     0                     1
     2                     0                     2
     3                     1                     5
     4                     0                     3
     5                     1                     6
     6                     0                     4
     7                     1                     7
```

Figure 4.39: **Affinity statistics for a nested parallel region using OMP_PLACES="sockets(1)"** – The initial mapping is the same as in the case where the explicit place list `PL_1` was used. In both examples, the first-level master thread was bound to hardware thread 0. The details are somewhat different, but to the application, this should not matter and the final mapping to the hardware threads is the same.

At this second level, the `close` policy is in effect. There are four threads ($T = 4$) and one place ($P = 1$). Following the placement algorithm for the "$T > P$" case, there is only one subset of threads (ST) per team and each contains all four threads in the respective team. It also follows from the algorithm that the three additional threads in each team are scheduled on the same socket where their master thread is running.

The runtime results in Figure 4.40 show that the application started on hardware thread 138, but migrated to hardware thread 128. As mentioned before, there is no notion of time in these tables, but with the full output (not shown here), the following thread creation and migration was reconstructed.

While executing on hardware thread 128, thread 0 created OpenMP thread 1. This thread was moved to hardware thread 0. At this point, the two threads execute

Thread	On HW Thread	Node	Created Thread
0	128	2	1
0	128	2	3
0	128	2	5
0	128	2	6
1	0	1	2
1	0	1	4
1	0	1	7

Thread	Running on HW Thread	Bound to HW Thread
0	138	128
1	128	0
2	0	2
3	128	129
4	0	1
5	128	130
6	128	131
7	0	3

Figure 4.40: **Affinity statistics for a nested parallel region using OMP_PLACES="sockets(2)"** – The other settings are the same as for the previous case. The two first-level OpenMP threads are scheduled onto a different socket. Each of these threads creates a team, and the threads in the team execute on a hardware thread in the same core as their master.

the outer-level parallel region. Both encounter the second-level region and act as the master thread for the teams created.

The tables show that thread 0 created threads 3, 5, and 6. These subsequently moved to hardware threads 129, 130, and 131. The four hardware threads $128, \ldots, 131$ are on socket 1. OpenMP thread 1 created threads 2, 4, and 7 on hardware thread 0. These migrated to hardware threads 2, 1, and 3 respectively.

The hardware thread numbers and the numbering in Figure 4.6 on page 163, confirm that the two outer threads are spread over the sockets and that their teams are executing on consecutive hardware threads within a core. They are truly "close."

Thread mappings for Case III In the last example of this section, it is shown how to use the abstract names to guarantee each OpenMP thread executes on its own core.

This is achieved by setting `OMP_PLACES=cores`. On the target two-socket system, this expands to a place list with 32 elements ($P = 32$).

As before, the `spread` policy applies to the first-level parallel region. There are two threads ($T = 2$), and the "$T \leq P$" algorithm applies. This results in 2 subpartitions, both with $P/T = 32/2 = 16$ consecutive places. Each place spans a full core.

OpenMP thread 0 executes in the subpartition where its master thread runs. Thread 1 executes in the other subpartition.

There are two sockets with 16 cores each and two subpartitions both with 16 places. One may expect each subpartition to exactly span a socket, but *there is no guarantee*. The assignment of places/cores to subpartitions is implementation-defined.[17]

Each of the OpenMP threads executes in a separate core, and they are at a distance of 16 in terms of resource numbers.

At the second level, the `close` policy is in effect for both teams. The subpartition of the respective master thread serves as the place list. There are 16 places in each list ($P = 16$) and four threads ($T = 4$). The algorithm for "$T \leq P$" applies and each OpenMP thread is assigned to a unique place, which is a core in this case.

With this, the goal has been achieved. All eight threads in the nested parallel region execute on their own core.

The runtime results in Figure 4.41 show that OpenMP thread 0 starts on hardware thread 66 and migrates to its destination thread (64). There, it creates OpenMP thread 1.

Thread 0 also creates the other threads in its team. In this case, threads 3, 5, and 7 are created. These threads start on the same hardware thread that their master thread is running on (64) and from there migrate to their destination hardware threads: 72, 80, and 88.

After thread 1 migrated from hardware thread 64 to 192, it created OpenMP threads 2, 4, and 6 on hardware thread 192. These migrated to hardware threads 200, 208, and 216 respectively.

[17] It is reasonable to expect that an implementation makes sensible choices in this case, though.

```
Thread   On HW Thread   Node   Created Thread
     0             64      1                1
     0             64      1                3
     0             64      1                5
     0             64      1                7
     1            192      2                2
     1            192      2                4
     1            192      2                6

Thread  Running on HW Thread    Bound to HW Thread
     0                    66                    64
     1                    64                   192
     2                   192                   200
     3                    64                    72
     4                   192                   208
     5                    64                    80
     6                   192                   216
     7                    64                    88
```

Figure 4.41: **Affinity statistics for a nested parallel region using OMP_PLACES=cores** – The other settings are the same as before. The eight OpenMP threads are distributed over the two sockets. On each socket, four cores execute the four OpenMP threads from the second-level parallel region.

The placement of the OpenMP threads confirms that the implementation spreads the two outer-level threads over the two sockets. The other three threads in each team run on the same socket as their master thread, and the cores used are next to each other. The specific mapping onto the sockets and cores is an implementation-dependent feature, however.

4.8.4 Moving Threads Around

So far, thread placement for a single (nested) parallel region has been the only consideration. In this section we show what happens when there are two separate single-level parallel regions with a *different* affinity policy.

The code fragment is shown in Figure 4.42. There are two single-level parallel regions. On the first region, the `proc_bind` clause is used to set the policy to `spread`. This is changed to the `close` policy on the second region.

```
 1 #pragma omp parallel proc_bind(spread)
 2 {
 3    #pragma omp single
 4    {
 5       printf("First single region - spread policy\n");
 6    } // End of single region
 7 } // End of parallel region 1
 8
 9 #pragma omp parallel proc_bind(close)
10 {
11    #pragma omp single
12    {
13       printf("Second single region - close policy\n");
14    } // End of single region
15 } // End of parallel region 2
```

Figure 4.42: **Two parallel regions with different affinity settings** – There are two single-level parallel regions. The first region uses the `spread` affinity policy, while the second region sets it to `close`. This causes the threads to migrate in between the execution of the two regions.

The number of threads is set to four for both regions (`OMP_NUM_THREADS=4`) and the place list is defined using the `cores` abstract name: `OMP_PLACES=cores`. The policies are activated at runtime by setting `OMP_PROC_BIND=true`.

The place list contains 32 places ($P = 32$), which is the number of cores in this system. There are four threads ($T = 4$), and for the first parallel region the "$T \leq P$" algorithm applies. There are four subpartitions with $P/T = 8$ consecutive places. Each thread is assigned to one of these subpartitions: $T_i \in SP_i$ for $i = 0, \ldots, 3$.

Although not guaranteed, it is reasonable to assume that the subpartitions are defined as shown in Figure 4.43. There are eight places per subpartition and each place spans the hardware threads within a single core. If we denote the set of hardware thread numbers in core i by C_i, the following holds: $C_i = \{8 * i, 8 * i + 1, \ldots, 8 * i + 7\}$ for $i = 0, \ldots, 31$. For example, $C_2 = \{16, \ldots, 23\}$.

Sub-partition	**All 32 places (each spanning one core)**			
SP_0	$\{0,\ldots,7\}$	$\{8,\ldots,15\}$	$\{\mathbf{16},\ldots,23\}$	$\{24,\ldots,31\}$
	$\{32,\ldots,39\}$	$\{40,\ldots,47\}$	$\{48,\ldots,55\}$	$\{56,\ldots,63\}$
SP_1	$\{64,\ldots,71\}$	$\{72,\ldots,79\}$	$\{\mathbf{80},\ldots,87\}$	$\{88,\ldots,95\}$
	$\{96,\ldots,103\}$	$\{104,\ldots,111\}$	$\{112,\ldots,119\}$	$\{120,\ldots,127\}$
SP_2	$\{128,\ldots,135\}$	$\{136,\ldots,143\}$	$\{\mathbf{144},\ldots,151\}$	$\{152,\ldots,159\}$
	$\{160,\ldots,167\}$	$\{168,\ldots,175\}$	$\{176,\ldots,183\}$	$\{184,\ldots,191\}$
SP_3	$\{192,\ldots,199\}$	$\{200,\ldots,207\}$	$\{\mathbf{208},\ldots,215\}$	$\{\mathbf{216},\ldots,223\}$
	$\{\mathbf{224},\ldots,231\}$	$\{\mathbf{232},\ldots,239\}$	$\{240,\ldots,247\}$	$\{248,\ldots,255\}$

Figure 4.43: **Places definitions for four threads and the cores place list** – The table shows in which way the hardware thread numbers map onto all 32 places when using four OpenMP threads and the `cores` place list. Each place spans the hardware threads within one core. The first column identifies the four subpartitions. The hardware threads used during execution of the example code are marked in boldface font.

The second parallel region uses the `close` affinity policy. There are again $T = 4$ threads and $P = 32$ places. These are all shown in Figure 4.43, including the hardware thread numbers for each place. Following the algorithm for the "$T \leq P$" case, the threads are assigned consecutive places, starting with the place where the master thread executes.

The first table in Figure 4.44 shows that OpenMP thread 0 executed on hardware thread 208 when it created the other three threads. Although there are two parallel regions, the threads are created only once and re-used for the second parallel region.

The first line in the second table shows that thread 0 migrated from hardware thread 212 to hardware thread 208 and then stayed there. After the three OpenMP threads 1, 2, and 3 were created there, they migrated twice. This is why there are two lines for each of these threads.

Initially these threads moved to hardware threads 16, 80, and 144, respectively. They are on cores C_2, C_{10}, C_{18}, and C_{26} respectively. The threads are evenly spread: the "distance" between consecutive cores is always 8.

```
Thread   On HW Thread   Node   Created Thread
     0            208      2                1
     0            208      2                2
     0            208      2                3

Thread  Running on HW Thread    Bound to HW Thread
     0                   212                   208
     1                   208                    16
     1                    16                   216
     2                   208                    80
     2                    80                   224
     3                   208                   144
     3                   144                   232
```

Figure 4.44: **Affinity statistics for the two parallel regions using the cores place list** – The results confirm that the threads are first spread, executing on hardware threads 208, 16, 80, and 144. On the next parallel region, the master thread remains on hardware thread 208, while the other three threads migrate to hardware threads 216, 224, and 232.

On the second parallel region, the master thread continued to run on hardware thread 208, but the other threads migrated to hardware threads 216, 224, and 232 on cores C_{26}, C_{27}, C_{28}, and C_{29}, respectively. These have a distance of 1. This is the `close` affinity policy in action.

4.9 Concluding Remarks

Thread affinity support allows the user to completely control where in the system the OpenMP threads should run. This is not a static placement and threads may move around in the system.[18]

The key concepts are the *OpenMP places* and the *affinity policies*. The latter defines the desired proximity (`master`, `close`, or `spread`) of the threads. This may be used to match the resource requirements of the application.

The places are used to control the scheduling granularity. When setting up the places and the place list, the user has the choice of either having full control using the resource numbers, or leveraging the symbolic names `sockets`, `cores`, or `threads`.

[18] Not all systems support thread migration. It is good practice to check this.

Both come with their pros and cons. The resource numbers are precise. There are no uncertainties as to where the threads execute, but the numbers depend on the OS, as well as the system characteristics. Most likely, they would need to be adapted when a different OS, or system is used. This is in sharp contrast with the symbolic names. They are guaranteed to be supported and provide portability, but a possible downside is that there may be some room for implementation-dependent behavior in the thread placement.

Below are several tips and tricks on how to make best use of this feature and to reduce the risk of making mistakes:

- A first step to optimize for performance on a cc-NUMA system is to use the First Touch placement policy to distribute the data in such a way that the threads mostly access their data from a nearby memory. Parallel data initializations are often sufficient to achieve this.
- If First Touch cannot be leveraged, for example when reading data, a redundant parallel initialization may help to distribute the data and avoid a single memory to become the bottleneck.
- If none of this is applicable, it is worth checking the documentation for a system call, for example `madvise()`, to specify a different placement policy.
- The `OMP_PLACES` and `OMP_PROC_BIND` environment variables have default settings. We strongly recommend setting these explicitly, or checking the documentation for the defaults.
- The use of environment variable `OMP_DISPLAY_ENV` is strongly recommended to verify the affinity settings.
- The place list is static. It is defined upon program startup and cannot be modified.
- Threads are not allowed to migrate between places.
- If a place contains multiple resource numbers, all numbers in the list are equal from a scheduling point of view.
- The affinity policy may be adapted on each parallel region.
- The abstract names are preferred when specifying the place list.

- An implementation may support additional abstract names to support specific architectural features.
- When more control over the placement is needed, the interval notation provides for a compact way to define the places. This is not only less error-prone, but also easier to adapt to other platforms.