Glossary

Address space The set of all legal addresses in memory for a process. It constitutes the amount of memory available to it. The OpenMP programming model assumes a shared address space for all threads in a process.

API Application Programming Interface. An API consists of a well-defined set of language features, library routines, annotations, or directives that may be employed by a programmer to solve a programming problem, often in a system-independent manner. An API often serves to hide lower-level implementation details from the user. OpenMP is an API for shared memory parallel programming.

ARB Architecture Review Board. An organization to maintain specifications and continue to evolve these to follow trends in the market.

Array section A subset of the elements of an array. Pointer-based array sections describe the size of pointed to memory. A zero-length pointer-based array section is used in `map` clauses on `target` constructs to find the corresponding accelerator address of the original pointed-to host address.

Atomic operation As suggested by the name, an atomic operation cannot be subdivided into smaller operations and it is uninterruptible, that is, only one thread at a time can execute such an operation.

An example is the "atomic add" which performs an operation of the type `a_val += new_value`. Atomic operations are used to correctly read and write shared variables.

They have been introduced to take advantage of highly efficient atomic instructions many processors support. OpenMP uses the `atomic` construct to support atomic operations.

Atomic read This guarantees that the read (or better, memory load instruction) of a variable is an atomic operation. It means that the value cannot change while the read/load is in progress.

Atomic write This guarantees that the write (or better, memory store instruction) of a variable is an atomic operation. It means that the value cannot change while the write/store is in progress.

Bandwidth For memory system transfer rates, the peak speed expressed in the number of bytes that can be transferred per second. There are different bandwidth rates between different parts of the memory system. For example, the transfer rate between a cache and CPU may be higher than the bandwidth between main memory and the cache. There may be multiple caches and paths to memory with different rates.

Barrier A synchronization point in the code where all threads in a team have to wait. No thread reaching the barrier can continue until all threads have reached the barrier. OpenMP provides explicit barriers as well as implicit barriers at the end of parallel loops, sections, single constructs, and parallel regions. The user has the option of removing barriers at the end of work-sharing constructs to achieve better performance. Barriers at the end of parallel regions cannot be removed.

Cache A relatively small, very high speed memory buffer between main memory and the processor. A cache contains data, instructions or both. In case of the latter it is called a unified cache. Data (or instructions) is copied from main memory to cache and back in blocks of contiguous data called a "cache line." The size of the line depends on the machine. A line may be evicted from cache if another data block needs to be stored in the same line. The strategy for replacing data in cache is system-dependent. Since cache is usually built from very high speed, but expensive memory components, it is substantially smaller than main memory. However, cache sizes tend to increase over time and on some platforms, small data sets might fit entirely into cache. Without cache, a program would spend the vast majority of its execution time waiting for the arrival of data, since processor speeds are significantly faster than memory access times. Computers may have several levels of cache, with those closest to the CPU being smaller, faster, and more expensive than those closer to main memory. The cache with the shortest access time is usually called an "L1 cache." The one with the second shortest access time is referred to as an "L2 cache," etc. Data is copied between levels. A different type of cache is called "Translation Lookaside Buffer" (TLB). It contains addresses mapping information.

Cache coherence The ability of a multi-processor (or multi-core) system to maintain the integrity of data stored in local caches of shared memory. On uniprocessor systems, data written into cache typically remains there until the cache line is replaced, at which point it is written back to a higher level cache or main memory. Multiprocessor systems share memory and caches. Without a mechanism for maintaining coherency between memory and cache, a load instruction executing on one processor might not have access to a value recently updated by other processors. With cache coherence, a processor is informed that data it needs is in a cache line that is stale or invalid, requiring a fetch of the updated value. The cache coherence protocol is architecture specific and implemented in various ways on different systems.

Cache line A level of cache is subdivided into lines, each of which may be filled by a block of data from main memory. The cache line size thus determines the unit of data transfer between the cache and memory, as well between the various levels of a specific cache type. The size of the line is system dependent and may differ from one level to another. For instance, a level-1 cache line could be smaller than a level-2 cache line. There is no unique best size for a cache line. A long line, for example, increases bandwidth but also increases the chance that false sharing occurs.

cc-NUMA cache coherent Non-Uniform Memory Access. These are NUMA systems where coherence is supported across all caches of the individual cores in the system. Maintaining cache coherence across shared memory is costly, but systems without this feature are extremely difficult to program. Therefore, shared memory NUMA systems available today generally provide some form of cache coherence and are really cc-NUMA systems. Thanks to cache coherence, data access is transparent throughout the entire system, but just as with a NUMA system, there is a notion of "local" and "remote" memory. For good scalability one had better take this into account and make sure a thread primarily accesses data in its local memory. Note that the existence of cache coherence does not prevent the user from introducing race conditions into OpenMP programs as a result of faulty programming.

Ceiling function The description can be found under "Floor function."

Compiler directive A source-code comment in Fortran or pragma in C/C++ that provides the compiler with additional information and guidance. In OpenMP, directives are used to identify parts of the code the compiler should parallelize, define shared and private data, thread synchronization, and more.

Composite construct A construct that is composed of two other constructs and has functionality that cannot be expressed by nesting one of the constructs immediately inside the other construct.

Computing the number of threads per subset With thread affinity support, threads are distributed over OpenMP places. If the number of threads T and the number of places P do not evenly divide, the distribution of threads over the sets of threads, the number of threads per place, or the number of places per subpartition, is not equal. Below we show the recipe how to compute this for one specific case. It is trivial to apply this to other scenarios.

Assume there are $T = 7$ threads and $P = 4$ places. If we use the `close` affinity policy, there are five sets with threads, but some sets contain $floor(7/4) = 1$ thread, while the remaining sets contain $ceiling(7/4) = 2$ threads. If we define the number of sets with one or two threads to be N_1 and N_2 respectively, the question is what their values are.

This problem is simply expressed in terms of two linear equations and then solved: $1 * N_1 + 2 * N_2 = 7$ and $N_1 + N_2 = 4$. It is easy to see that $N_1 = 1$ and $N_2 = 3$ is the unique solution.

In other words, there is one set with 1 thread and three sets contain 2 threads.

Conditional compilation This is a Fortran-only feature to deal with OpenMP specific run time functions in case the source is not compiled with the recognition of OpenMP directives enabled, or if the compiler does not support

OpenMP. Without special precautions, usage of OpenMP run time functions leads to unresolved references when linking the object files. The solution is to start the source line that uses the run time function with `!$` (or `C$/*$` in case old style fixed format Fortran is used). The specifications guarantee that these two characters are replaced by two spaces in case OpenMP is supported and enabled at compile time. This means the comment line is expanded to an executable statement, but only in an OpenMP context. For more details we refer to the specifications. An example can be found in [2].

Contention group An initial thread and all its descendent threads. An initial thread is not a descendent of another initial thread.

Core A component of a microprocessor that is able to load/store data and execute instructions. Beyond this generic description there are significant implementation differences. Depending on the architecture, hardware resources, and components may be shared among, or specific to, a core. Examples are the register file, caches (for both data and instructions), functional units, and paths to other components. In some designs, an individual core is able to execute multiple independent threads.

CPU Central Processing Unit. The CPU is also referred to as the "processor." The circuitry within a computer that executes the instructions of a program and processes the data. Current designs often contain multiple cores.

Data race A data race occurs if multiple threads simultaneously modify the same (and therefore shared) memory location. The outcome of such an update is undetermined and most likely varies from run to run. The result is also affected by the number of threads used. After a data race has occurred, the behavior of the program is undefined. Note that by definition, a data race also causes false sharing, usually resulting in a performance impact. There are ways to avoid data races. The critical section is probably the most commonly used OpenMP construct to prevent data races from happening.

Deadlock A situation where one or more threads are waiting for a resource that will never become available. There are various ways to introduce such deadlock situations into an OpenMP application. For example, a barrier inserted at a point not encountered by all threads, or improper calls to OpenMP locking routines, will result in deadlocks.

Device A processing element that executes program code. A device has access to a memory that may or may not be shared with other devices.

Device data environment Referred to in the context of an accelerator, the set of variables and ICVs that are available to a device. It includes the set of variables that are currently mapped to an accelerator. A variable is present on the accelerator if it is in its device data environment

Device pointer A pointer variable on the host device whose value is an object that represents an address in an accelerator's address space.

Directive sentinel A special sequence of characters that indicates that the line is to be interpreted as a compiler directive. OpenMP directives in Fortran must begin with a directive sentinel. The format of the sentinel differs between fixed and free-form source files.

Doacross loop A loop-carried dependence inhibits straightforward parallelization of a loop. In case of a doacross loop however, code transformations and/or additional synchronization(s) result in a loop that can be parallelized after all. The run time benefit depends on the details because sometimes an additional overhead is introduced, or the parallelism is limited even after the transformation(s).

DSP Digital Signal Processor. It is an example of a hardware accelerator. DSPs can be found in many embedded devices, cell phones being a case in point. They are specialized to efficiently execute signal, vision and image processing applications.

DTrace DTrace stands for "Dynamic Tracing." Initially available on the Oracle Solaris^{TM}operating system, it has since then been ported to Mac OS X^{TM}and Oracle Linux^{TM}.

A full description of DTrace as well as examples can be found at [7]. There is also a DTrace website [6] and several books cover this tool in great detail [17, 11, 10].

DTrace is a tool that allows one to trace activities in the operating system kernel, as well as user applications if they have been instrumented for this. The main use is to follow what happens in the kernel though. A very powerful feature is that the kernel is ready to be instrumented at any time, but if instrumentation is not requested, there is no overhead at all. There is also no need to use a special kernel. DTrace is there and ready to go.

Although one can use DTrace at the command line, most users prefer to write a script to do this. DTrace comes with its own language "D." The syntax is very easy to learn for those with programming experiences in C/C++, awk, perl, etc.

In Chapter 4 starting on 151 it is shown how DTrace is used to verify the thread affinity policies and see where threads end up running in the system.

There is a related tool for Linux. It is called SystemTap [25].

Environment variable A Unix shell variable that may be set by the user to influence some aspect of a program's execution behavior. The OpenMP API includes a set of environment variables to control aspects of the program's execution, including the number of threads to use, the default work schedule, and the use of nested parallelism.

False sharing A situation where multiple threads update the same cache line. Since the granularity of information used by cache coherence mechanisms is a cache line, the system cannot distinguish whether there is true sharing, or if independent updates happen to be on different chunks of the same cache line. False sharing results in poor performance, but it is not an error affecting correctness of the numerical results.

First touch This is a common page placement policy and the default on most, if not all, Operating Systems. Under this policy, the thread that touches the data first, owns the data. Technically, this is controlled through the D-TLB. The first time an entry in this cache is created, ownership and location are determined. With this policy, the page resides in the memory connected to the socket where the thread executes. Unless there is a capacity issue and some data needs to be placed elsewhere.

Floor function This mathematical function maps a floating point number to an integer. For a floating point number x, $floor(x)$, or $\lfloor x \rfloor$, returns the largest integer not exceeding x. For example, $floor(2.3) = \lfloor 2.3 \rfloor = 2$.

A related function is the ceiling function: $ceiling(x)$, or $\lceil x \rceil$, returns the smallest integer not less than x. For example, $ceiling(2.3) = \lceil 2.3 \rceil = 3$.

Generating task A task that generates another task.

Globally linked variable A variable that is linked to all accelerators for the whole program. A globally linked variable can be accessed in a mapped function as long as it is present in the device data environment of the accelerator where the mapped function is being executed.

Globally mapped variable A variable that is mapped to all accelerators for the whole program. A globally mapped variable has an infinite reference count, is never removed from a device data environment, and is initialized on all devices when a program starts.

GPGPU General Purpose Graphics Processing Unit. Nowadays the term "GPU" is more commonly used. See the description for GPU.

GPU Graphics Processing Unit. A GPU is an accelerator for graphics operations. The original GPU was extremely fast on specific (graphics) operations only, but this is no longer the case. The strength of a GPU lies in the massive parallelism it offers. Each node in the GPU is not very powerful, but there are many of such nodes. GPUs can be used for all sorts of floating-point intensive computations. An issue with this type of hardware is the programming model. Since the GPU is connected to the main processor it is an example of a hardware accelerator, but with a different instruction set, memory architecture, etc. Various programming models are available for GPUs.

Hardware thread To start with, there are many different implementations of this concept. The differences can be substantial and the description below is a generalization.

A *hardware thread*, or "strand," is not one entity. It is the combination of one or more pipelines (that execute the instructions) and "state."

With "state" we mean additional hardware resources to execute the instructions coming from a software thread, or process. For example, there could be multiple Register Files (RF). When a software thread starts executing instructions in the pipeline it gets assigned one of these register files and while executing it has its own dedicated RF. This is a good thing because other threads executing cannot interfere with its register values. Another example is an Instruction Buffer (IB) for each hardware thread.

In general, the more dedicated hardware resources (a richer "state"), the more efficient the hardware threads execute.

Nowadays, many cores have this kind of dedicated hardware support to efficiently execute multiple threads within the same pipeline(s). It is as if each instruction executing in the pipeline is tagged with an identifier to distinguish the various threads. Although there are significant implementation details, at a sufficiently high level it appears as if multiple threads are executing simultaneously. This is why this kind of technology is known under various other names like "Virtual threading" or "Simultaneous threading."

Home node The home node is the location where the data resides in memory. On a flat memory system, this is irrelevant, but on a cc-NUMA architecture it is an important concept. All pages in the system have a home node. The placement policy dictates where this is. See also the entry for *First Touch*.

Host device The initial device where program execution begins.

Host fall back The concept that a device construct may fall back and execute on the host device.

ICV Internal Control Variable. This is OpenMP specific terminology for certain variables that are used to store information used by the run time system. For example, the number of threads is stored in ICV `nthreads-var`. Prior to program execution, this variable is set to the default number of threads used, which is implementation defined. At run time, this value may change. The application can never access ICVs directly, but through run time functions the value can be read and possibly modified.

Idle thread A software thread that is not performing any work is said to be *idle*. By default the Operating System decides what to do with such a thread. Keeping it busy ("spinning") in the core is usually good for performance, but wastes cycles. Putting it to sleep frees up resources for other threads, but waking up a sleeping thread is relatively costly Through environment variable `OMP_WAIT_POLICY` an OpenMP can specify a preference for this.

Initial thread A thread that starts the execution of an OpenMP program. A thread that starts the execution of a target region. One of the threads in a league that, in parallel, start the execution of a target teams region. An initial thread starts a contention group.

Latency This is the time spent waiting for a response. Memory latency is the time it takes for data to arrive after the initiation of a memory reference. A data path with high memory latency would be inappropriate for moving small amounts of data.

LCD See Loop-carried dependence.

League The set of initial threads, each in their own team, that is started by a `teams` construct. Each team is a contention group.

Lexical If used in the context of (parallel) programming, the informal way to describe this is that a construct, or other source code element, is directly visible in the program source. For example, "tasks can be lexically nested" means that a tasking directive can be included within another tasking directive.

Lock A mechanism for controlling access to a resource in an environment where there are multiple threads of execution. Locks are commonly used when multiple threads need to perform some activity, but only one at a time is allowed to do so. Initially, all threads contend for the lock. One thread gets exclusive access to the lock and performs the task. Meanwhile, the other threads wait for the lock to be released. When that happens, a next thread takes ownership of the lock, and so forth. There may be a special data type for declaring locks. In a C/C++ OpenMP program, this is the `omp_lock_t` type. In Fortran, it must be an integer variable of `kind=omp_lock_kind`.

Lock contention In case multiple threads access the same lock simultaneously, or very shortly after each other, the lock is said to be "contended." Such a situation can greatly degrade parallel performance. The cache line containing the lock variable needs to be acquired by the threads and quickly this can become the bottleneck.

One approach to make things go smoother is to use a backoff algorithm. If a thread fails to obtain the lock, it waits for a while. Upon successive failures to acquire the lock, this delay time increases either linearly, or exponentially. This often helps to improve the performance, but if the lock is not so heavily contended, may take somewhat more time.

This is why OpenMP supports a hint to be specified when initializing the lock variable. Through this hint, the developer can indicate whether a lock is contended, or not, for example.

Loop overhead This is the cost associated with the execution of the instructions related to the mechanics of executing a loop. For example, the loop iteration variable needs to be incremented and a branch instruction is required to either terminate the loop or proceed to the next iteration. All such instructions are collectively called the "loop overhead." For a short loop with little work performed per iteration this cost can dominate the total execution time. On longer loops, or more work per iteration, this cost is less important.

Loop-carried dependence This is a loop where the result for a loop iteration depends on an earlier iteration, or iterations. For example, a statement like this has a loop-carried dependence: `a[i] = a[i-1] + 1`. For a given loop iteration `i`, result `a[i-1]` has to be computed first, before `a[i]` can be computed.

Mapped function A function that can be called from code running on an accelerator.

Mapped variable A variable that is shared between a host device and one or more accelerator devices. A mapped variable must have a type that is bitwise copyable. Pointer variables may be mapped, but what they point to is not mapped.

The original variable on the host is mapped to a corresponding variable in the accelerator's device data environment. Depending on the underlying memory system, the original and corresponding variable may or may not share the same memory location. An original variable may have only one corresponding variable in an accelerator's device data environment.

Mapped variables have map-enter and map-exit phases that occur at the entry to or exit from a device construct that maps the variable. A corresponding variable has a reference count that is incremented in the map-exit phase and decremented in the map-enter phase.

A mapped variable has a map-type that determines how the consistency of the original and corresponding variables is managed. The map-type is used to minimize the performance overhead of copying values between a host device and an accelerator.

Mflop/s This is an abbreviation of “million floating-point operations per second.” Mflop/s is a performance metric that can be calculated if the number of floating-point operations (flops) is known. Dividing flops by the execution time in seconds and scaling by 10^6 gives Mflop/s. Related metrics are Gflop/s (gigaflop/s), Tflop/s (teraflop/s) and Pflop/s (petaflop/s).

MPI Message Passing Interface. This is a de facto standard API that was developed to facilitate programming for distributed-memory architectures. MPI consists of a collection of library routines. In an MPI program, multiple processes operate independently and communicate data via messages that are inserted by the programmer. The MPI API specifies the syntax of the functions and format of the messages. MPI is increasingly mixed with OpenMP to create a program that is tailored to exploit both the distributed memory and shared memory, as supported in a cluster of shared-memory nodes. This model can also be used to take advantage of hierarchical parallelism in an application.

MPI Forum An open group with representatives from many organizations that define and maintain the MPI standard. The official website is at [18].

Multi-core A microprocessor design with multiple cores integrated onto a single processor die. Just as with the individual cores, there are vast differences between the various designs available on the market. Among others, there are differences in cache access and organization, system interface, and support for executing more than one independent thread per core.

NUMA Non-Uniform Memory Access. What it means is that there is a notion of “local” versus “remote” memory. This is because a processor is only directly connected to a portion of the total memory, but through an interconnect can also access the memory connected to other processors. Typically, cores that are part of the same processor have the same access time to their own memory, but a longer access time to a memory further away. Historically, the word “NUMA” was reserved for clustered systems connected through a conventional network like Ethernet or Infiniband. Neither of these support cache coherence and the cost of accessing memory on another node in the network is relatively costly. Depending on the topology of the network, the remote cost need not be the same either. A memory can either be close by ("one hop"), or further away ("multiple hops"). The typical way to program such systems was, and still is, with a distributed memory programming model like MPI. If the interconnect supports cache coherence, a system consisting of multiple compute nodes is referred to as "cc-NUMA." Although there is a distinct difference in terms of performance and ease of use with a NUMA cluster architecture, nowadays, *NUMA* and *cc-NUMA* are often used interchangeably, but in practically all cases the latter is meant.

OpenMP ARB The ARB was created to maintain the OpenMP specifications and keep OpenMP relevant to evolving computer architectures, programming languages, and multiprocessing paradigms. Members of this ARB are organizations, not individuals. The website is at http://www.openmp.org.

OpenMP region This is all code encountered during a specific instance of the execution of a given OpenMP construct or library routine. A region includes any code in called routines, as well as any implicit code introduced by the OpenMP implementation.

Operating system page The Operating System (or OS for short) manages data at the *page* level. A page is a relatively large block of memory. The typical size for a default page size is 4 or 8 Kbyte. Even if an application uses a single byte element only, the OS allocates a page to contain this element. All actions on pages are under control of the OS. It is at the page level that data is placed and possibly moved around. As one can imagine, conflicts may arise if on a cc-NUMA system multiple threads execute in a different socket, but access the same page. This degrades performance and should be avoided. This is why it is best to allocate per-thread data on page boundaries.

There is a link with the TLB. Each page needs to have a mapping in the TLB before it can be used. In case only a small fraction of the data in a page is used, more TLB misses than necessary may occur. In such a situation, changing the page size (if supported of course) can improve performance.

Parallel overhead Parallel execution comes with a cost. A sequential program needs to be transformed into a parallel program. Typically this is achieved by inserting calls to an underlying parallel run time system. This system performs a variety of tasks, including the creation and management of the threads, but it may also need to handle data transfer, communication and synchronization of the threads. These are all activities one does not have in a serial program. The cost of all of this is usually referred to as "parallel overhead."

Parallel programming model A set of rules and features to express and control parallelism in an application. Examples are Posix Threads, Java Threads, OpenMP, MPI, and OpenCL.

Parallel scalability The behavior of an application when an increasing number of threads are used to solve a problem of constant size. Ideally, increasing the number of threads from 1 to P yields a parallel speed-up of P.

Parallel speed-up The ratio of the wall-clock time measured for the execution of the program by one thread to the wall-clock time measured for execution by multiple threads. Theoretically, a program run on P threads should run P times as fast as the same program executed on only one thread.

Perfectly nested loop In a perfectly nested loop the individual loops are back to back. That is, there is no other code in between the loop statements.

Pointer aliasing A pointer in an application points to an address in memory. A pointer alias exists if different pointers share the same memory address. If this is the case, a change made through one pointer, affects the value pointed to by another pointer. Programming languages all have their own default rules in how far aliasing is (not) allowed. Pointer aliasing can have a profound effect on the performance. The number one goal of any compiler is to produce correct results. Pointer aliasing may force the compiler to generate slower code than necessary. If the compiler knows that different pointers cannot be aliased, much more efficient code may be generated. This is why the C99 `restrict` keyword is important to consider and apply where relevant. Compilers also often support various options to specify the extent of the aliasing.

Process An entity created by the Operating System to execute an application. A process has its own state information code and data. At runtime, it has its private set of registers, address space, program counter, and stack pointer.

A process can have multiple threads of control and instructions, which is the basis for the OpenMP programming model.

Processor A physical resource that may be used to execute programs. Many different processors have been built, some of which are designed for a special purpose. A general-purpose processor is typically able to execute multiple instructions simultaneously, because it has functional units that can operate independently. A conventional processor executes multiple processes via time slicing, in which each gets some share of the overall CPU time. To achieve this, the state of a program including the values in its registers, is saved and the state of another process loaded. This is known as context switching. Context switching is considerably faster for threads, as they share some of their state. The processes or threads appear to be executing in parallel. A single processor may also be able to execute multiple instruction streams simultaneously by interleaving their instructions or by permitting two or more threads to issue instructions. Machines with this capability appear to the user to be a shared memory parallel computer.

Race condition A programming fault that produces unpredictable program behavior due to unsynchronized concurrent executions. Race conditions are hard to find with conventional debugging methods and tools. Most common is the *data race condition* that occurs when two or more threads access the same shared variable simultaneously with at least one thread modifying its value. Without the appropriate synchronization to protect the update, the behavior will be indeterminate, and the results produced by the program will differ from run to run. Other, more general race conditions are also possible.

Register file The register file is part of a core. Its entries are called "registers." Values in registers are immediately accessible. Registers are what instructions use as their operands. For example, (pseudo) instruction `add %r1,%r2,%r3` adds two values stored in registers `%r1` and `%r2` respectively. The result is stored in register `%r3`. Data is fetched from memory and stored back through load and store instructions. These instructions use an address in memory and a destination register to load a value into or store from.

Scoping In an OpenMP program, each variable is of a certain memory type. It is basically either a private, or a shared variable. There are default rules for this data environment, but one can also explicitly specify the nature of a variable. This is often referred to as “scoping a variable.”

Semaphore A semaphore is a synchronization object used in parallel programming to control access to a shared resource. It can be used to arbitrate access and avoids conflicts between multiple threads.

Sequential consistency Defined by Leslie Lamport [16], a (shared memory) parallel program implementation conforms to this property if it guarantees that “the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.” If applied to an OpenMP program, this requires a memory update after each operation that modifies a shared variable and, potentially, before each use of a shared variable.

Side effect A side effect in the context of programming is said to occur if a variable changes as the result of a modification to a seemingly unrelated variable. This can happen because of pointer aliasing for example. In such a situation, two different pointers point to the same memory location. Changing one pointer affects the others. Another example is a function that modifies global data. Since global data is (potentially) used by other functions, this can wreak havoc in a parallel program and care needs to be taken to handle this situation.

SIMD Single Instruction Multiple Data. A SIMD instruction performs the same operation (like an addition) on multiple data elements and does so simultaneously. It is another example of parallel execution, but at a fairly low level. The instructions are typically generated by the compiler and works best on vector type of operations. This is why it is also called “micro-vectorization.”

SIMD instruction An instruction that operates on multiple data elements simultaneously, also referred to as vector instruction.

SMP Symmetric Multi-Processor. This is a computer system whose individual processors share memory in such a way that each of them can access any main memory location in the same amount of time. Until about a decade ago, many small shared-memory machines had an SMP architecture, while larger systems often had a cc-NUMA architecture. are symmetric in this sense, larger systems do not necessarily satisfy this definition. On such non-symmetric systems, the physical distance between a CPU and memory will determine the length of time it takes to retrieve data from memory. Memory access times do not affect the OpenMP programming model, but can have a significant impact on the performance of the application. The term can also refer to other kinds of shared-memory computers. In this book, we use the acronym SMP broadly to refer to all platforms where multiple processes share memory.

Socket This is a word that has become more popular with the advent of multi-core architectures. In the context of computer architecture, a "socket" is physical location on the motherboard of a computer system. It is where the chip with the processor is placed. The processor may or may not have multiple cores and optionally each core could have support for multiple hardware threads. The socket typically has connections to the rest of the system, including memory. In case of a cc-NUMA architecture, each socket is connected to a portion of the total memory available and the sockets are connected through a cache coherent interconnect.

Stack memory This is often abbreviated to "the stack." It is a region of memory that is used as a scratchpad when executing a function or subroutine. It is used to manage variables local to the function for example. Unless one writes assembly code directly, managing the stack is under control of the compiler. The Operating System defines a default size for the stack, but the user can change this. For example, on Unix systems with the Bourne or Bash shell, one can use the `ulimit` command. Check the documentation of your OS how to do this.

Unfortunately, OpenMP users get confronted with the stack more easily than others. This is because most, if not all, compilers use "outlining" that pushes regular code into a function body. Private variables are then local variables to the function, but that means stack space is needed to manage them. If not enough space is available, the program may crash and environment variable `OMP_STACKSIZE` needs to be used to increase the size of the stack for the threads.

Strand This word is often used by computer architects to denote what we call a "hardware thread" in this book. Please check this entry for a description.

Structured block For C/C++ programs, an executable statement with a single entry at the top and a single exit at the bottom. In Fortran code, it refers to a block of executable statements with a single entry at the top and a single exit at the bottom. An alternate name for this is *basic block*.

Synchronization Synchronization is used to coordinate the actions of multiple threads. It is essential in order to ensure correctness of the application. By default, an OpenMP program has barrier synchronization points at the end of parallel work-sharing constructs and parallel regions, where all threads have to wait until the last thread has finished its work. Synchronization may be expressed in many ways. OpenMP provides several constructs for explicit thread synchronization that should be used if accesses to shared data need to be ordered or if interference between multiple updates is to be avoided. These include critical regions, atomic updates, lock routines, and barriers. Memory synchronization, where thread-local shared data is made consistent with the process-wide values, is achieved via flushing.

Target task A task that is generated by a device construct. By default it is an included task, but if a `nowait` clause appears on the construct, it is a deferrable task.

Thread An Operating System entity that executes a stream of instructions. A process is executed by one or more threads and many of its resources (e.g., page tables, address space) are shared among these threads. However, a thread has some resources of its own, including a program counter and an associated stack. Since so few resources are involved, it is considerably faster to create a thread or to context switch between threads than it is to perform the same operation for processes. Sometimes threads are known as lightweight processes. In Unix environments, a thread is generally the smallest execution context.

Thread affinity This controls where in the system a thread should run and allows the application to leverage the system hardware characteristics. For example, if an application requires a significant amount of bandwidth, it may be beneficial to place the threads across the socket. If on the other hand, threads communicate very frequently, placing them on the same socket may be the best thing to do for performance. Often, thread affinity is combined with thread binding to re-schedule the thread on the same hardware thread, after a context switch.

Thread binding With thread binding, a thread is pinned down to a specific core or even hardware thread. If the Operating System supports this, the thread will never be rescheduled onto another core or hardware thread. This provides the optimal thread affinity and is supported in OpenMP. Be aware though that binding may slow down an application in case other applications are running as well, or if there are no hardware threads left for the OS to run on. In case of the latter, the OS will claim the resources it needs, potentially slowing down one or more application level threads.

Thread ID A means to identify a thread. In OpenMP the thread IDs are consecutive integer numbers. The sequence starts at zero, which is reserved for the master thread, and ends with $P-1$, if P threads are used. The `omp_get_thread_num()` function call enables a thread to obtain its thread ID.

Thread-safe A property that guarantees software will execute correctly when run on multiple threads simultaneously. Programs that are not thread safe can fail due to race conditions or deadlocks when run with multiple parallel threads. Particular care has to be taken when using library calls or shared objects and methods within OpenMP parallel regions.

TLB Translation Lookaside Buffer. The TLB buffer, or cache, is an important part of the memory system. It is a relatively small cache that maintains information on the physical pages of memory associated with a running process. If the address of data needed by a thread is loaded but not covered through the TLB, a TLB miss occurs. Setting up a new entry in the TLB is an expensive operation that should be avoided where possible.

Transactional memory With this, a lock is initially ignored and the code in the locked region is executed unconditionally. Only afterwards it is checked if there was a lock collision, or not. In case of the former, a rollback mechanism is triggered to ensure correct results. The advantage is that handling a lock that is not contended is really fast. If however, the lock is heavily contended, the cost of executing the rollback part can reduce, or even ruin, any performance benefit and ultimately handling the lock may be more expensive.

Several research and commercial implementations have become available in recent years. There are significant variations between the various architectures, including hybrid versions augmenting the hardware support with software features to enhance the efficiency.

Vector instruction An instruction that operates on multiple data elements simultaneously, also referred to as SIMD instruction.

Vector width The number of vector operations executed simultaneously in a vector, or SIMD, instruction.

Vectorization This is hardware acceleration first found in mainframes in the late 60's. Soon after, special vector machines were designed and built. The main idea is to perform the same basic operation, like an add, on a series of operands, the "vectors." Much later, this technology was implemented in micro-processors. Nowadays it is usually referred to as "SIMD." Refer to SIMD entry for more information.

Wall-clock time A measure of how much actual time it takes to complete a task, in this case a program or part of a program. Wall-clock time is also referred to as "elapsed time." It is an important metric for parallel programs. Although the aggregate CPU time most likely goes up, the wall-clock time of a parallel application should decrease when an increasing number of threads is used to execute it. Care needs to be taken when measuring this value. If there are more threads than processors or cores on the system, or if the load is such that a thread will not have a processor or core to itself, there may be little or no reduction in wall-clock time when adding a thread.