APPENDIX

Glossary

E

The specialized vocabulary used in this book is defined here. In some cases, where existing terminology is ambiguous, we have given all meanings but note which meaning we primarily use in this book.

absolute speedup: Speedup in which the best parallel solution to a problem is compared to the best serial solution to the same problem, even if they use different algorithms. See ***relative speedup***.

access controls: Any mechanism to regulate access to something, but for parallel programs this term generally applies to shared memory. The term is sometimes extended to I/O devices as well. For parallel programming, the objective is generally to provide deterministic results by preventing an object from being modified by multiple tasks simultaneously. Most often this is referred to as ***mutual exclusion***, which includes locks, mutexes, atomic operations, and ***transactional memory*** models. This may also require some control on reading access to prevent viewing of an object in a partially modified state.

actual parallelism: The number of physical ***workers*** available to execute a parallel program.

algorithmic skeleton: Synonym for ***pattern***, specifically the subclass of patterns having to do with algorithms.

algorithm strategy pattern: A class of patterns that emphasize the parallelization of the internal workings of algorithms.

aliasing: Refers to when two distinct program identifiers or expressions refer to overlapping memory locations. For example, if two pointers `p` and `q` point to the same location, then `p[0]` and `q[0]` are said to alias each other. The potential for aliasing can severely restrict a compiler's ability to optimize a program, even when there is no actual aliasing.

Amdahl's Law: Speedup is limited by the non-***parallelizable*** serial portion of the work. Compare with other attempts to characterize the bounds of parallelism: ***span complexity*** and ***Gustafson–Barsis' Law***. See Section 2.5.4.

application binary interface (ABI): A set of binary entry points corresponding to an ***application programming interface***. Fixed ABIs are useful to allow relinking to different implementations of a library module.

application programming interface (API): An interface (set of function calls, operators, variables, and/or classes) by which an application developer uses a module. The implementation details of a module are ideally hidden from the application developer and the functionality is only defined through the API.

arithmetic intensity: The ratio of computational (typically arithmetic) operations to communication, where communication includes memory operations. Comparing this ratio for an algorithm with the hardware's ratio gives a hint of whether computation or communication will be the limiting resource. See Section 10.3.

array operations: See ***vector operations***.

array processors: See ***vector processor***.

array-of-structures (AoS): A data layout for collections of heterogeneous data where all the data for each element is stored in adjacent physical locations, even if the data are of different types. Compare with ***structure-of-arrays***.

associative cache: A cache organization where copies of data in main memory can be stored anywhere in the cache.

associative operation: An operation $\otimes$ is associative if $(a \otimes b) \otimes c = a \otimes (b \otimes c)$. Modular integer arithmetic is associative. Real addition is associative, but floating-point addition is not. However, sometimes the roundoff differences from reassociating floating-point addition are small enough to be ignored, in which case floating-point operations can be considered approximately associative.

asymptotic complexity: Algebraic limit on behavior, including time and space but also ratios such as ***speedup and efficiency***. See ***big O notation, big Omega notation***, and ***big Theta notation***.

asymptotic efficiency: An ***asymptotic complexity*** measure for ***efficiency***.

asymptotic speedup: An ***asymptotic complexity*** measure for ***speedup***.

atomic operation: An operation guaranteed to appear as if it occurred indivisibly without interference from other threads. For example, a processor might provide a memory increment operation. This operation needs to read a value from memory, increment it, and write it back to memory. An atomic increment guarantees that the final memory value is the same as would have occurred if no other operations on that memory location were allowed to happen between the read and the write. See Section C.10, and ***lock*** and ***mutual exclusion***.

atomic scatter pattern: A ***non-deterministic*** data pattern in which multiple writers to a single storage location result in exactly one value being written and all others being discarded. The value written is chosen non-deterministically from the multiple sources. The only guarantee is that the resulting value in the target memory locations will be one of the values being written by at least one of the writers. See Section 6.2.

attached co-processor: A separate processor, often on an add-in card (such as a PCIe card), usually with its own physical memory, which may or may not be in a separate address space from the host processor. Often also known as an accelerator (although it may only accelerate specific workloads).

auto-vectorization: Automatically generating ***vectorized*** code from programs expressed using serial programming languages.

autotuning: The process of automatically adjusting parameters in parameterized code in order to achieve optimal performance.

available parallelism: See ***potential parallelism***.

bandwidth: The rate at which information is transferred, either from memory or over a communications channel. This term is used when the process being measured can be given a frequency-domain interpretation. When applied to computation, it can be seen as being equivalent to ***throughput***.

barrier: When a computation is broken into phases, it is often necessary to ensure that all threads complete all the work in one phase before any thread moves onto another phase. A barrier is a form of ***synchronization*** that ensures this. Threads arriving at a barrier wait there until the last thread arrives, then all threads continue. A barrier can be implemented using an ***atomic operation***. For example, all threads might try to increment a shared variable, then ***block*** if the value of that variable does not equal the number of threads that need to synchronize at the barrier. The last thread to arrive can then reset the barrier to zero and release all the blocked threads.

big O notation: Complexity notation that denotes an upper bound; written as $O(f(n))$. Big O notation is useful for classification of algorithm efficiency. In particular, big O notation is used to classify algorithms based on how they respond to changes in their input set in terms of processing time or other characteristics of interest.

For instance, a bubble sort routine may be described as taking $O(n^2)$ time because the time to run a bubble sort routine is proportional to the square of the size of the data set to sort. Since big O notation is about asymptotic growth, it may neglect significant constant factors. A pair of algorithms with running times of $n^2 + 100n + 10^{19}$ and $5n^2 + n + 2$, respectively, are both generally described as $O(n^2)$, despite significant differences in performance for most values of n.

For characterizing the suitability of an algorithm for parallel execution, ***big O*** analysis applies to both the ***work complexity*** and the ***span complexity***, but typically ***big Theta notation*** is preferred. See Section 2.5.7.

big Omega notation: Complexity notation that denotes a lower bound; written as $\Omega(f(N))$. See Section 2.5.7.

big Theta notation: Complexity notation that denotes an upper and a lower bound; written as $\Theta(f(N))$. See Section 2.5.7.

binning: The process of subdividing labeled data in a collection into separate sub-collections, each with a unique label. See ***bin pattern***.

bin pattern: A generalized version of the ***split pattern***, which is in turn a generalization of the ***pack pattern***, the bin pattern takes as input a collection of data and a collection of labels to go with every element of that collection, and reorganizes the data into a category (a bin) for every unique label in the input. The determinisitic version of this pattern is stable, in that it preserves the original order of the input collection. One major application of this pattern is in radix sort. It can also be used to implement the ***category reduction pattern***. See Section 6.4.

BLAS: The Basic Linear Algebra Subprograms are routines that provide standard building blocks for basic vector and matrix operations. The Level 1 BLAS perform scalar, vector, and vector–vector operations; the Level 2 BLAS perform matrix–vector operations; and the Level 3 BLAS perform matrix–matrix operations. Because the BLAS are efficient, portable, and widely available, they are commonly used in the development of high-quality linear algebra software (LAPACK, for example). A sophisticated and generic implementation of BLAS has been maintained for decades at `http://netlib.org/blas`. Vendor-specific implementations of BLAS are common, including the Intel Math Kernel Library (MKL), which is a highly efficient version of BLAS and other standard routines for Intel architecture.

block: Block can be used in two senses: (1) a state in which a thread is unable to proceed while it waits for some synchronization event, or (2) a region of memory. The second meaning is also used in the sense of dividing a loop into a set of parallel tasks of a suitable ***granularity***. To avoid confusion in this book, the term ***tile*** is generally used for the second meaning, and likewise the term ***tiling*** is preferred over "blocking."

branch and bound pattern: A ***non-deterministic*** pattern designated to find one satisfactory answer when many may be possible. Branch refers to using concurrency, and bound refers to limiting the computation in some manner—for example, by using an upper bound (perhaps the best result found so far). This pattern is often used to implement search, where it is highly effective. See Section 3.7.1.

burdened span: The ***span*** augmented with overhead costs.

by reference: A parameter to a function that acts exactly as if it were the original location passed to the function.

by value: A parameter to a function that is a copy of the original value passed to the function.

cache: A part of memory system that stores copies of data temporarily in a fast memory so that future uses for that data can be handled more quickly than if the request had to be fetched again from a more distant storage. Caches are generally automatic and are designed to enhance programs with ***temporal locality*** and/or ***spatial locality***. Caching systems in modern computers are usually multileveled.

cache coherence: A mechanism for keeping multiple copies of the same data in different caches consistent.

cache conflict: When multiple locations in memory are mapped to the same location in a cache only a subset of them can be kept in cache.

cache fusion: An optimization for a sequence of ***/vector operations/*** where the vector operations are broken into ***tiles*** and the entire sequence executed on each tile, so that the intermediate values can be kept in cache.

cache line: The units in which data retrieved and held by a ***cache***; in order to exploit spatial locality, they are generally larger than a word. The general trend is for increasing cache line sizes, which are generally large enough to hold at least two double-precision floating-point numbers, but unlikely to hold more than eight on any current design. Larger cache lines allow for more efficient bulk transfers from main memory but worsen certain issues, including ***false sharing*** which generally degrades performance.

cache-oblivious programming: Refers to designing an algorithm to have good cache behavior without knowing the size or design of the cache system in advance. This is usually accomplished by using recursive patterns of data locality so that locality is present at all scales. See Section 8.8, as well as [ABF05] and [Vit08].

cancellation: The ability to stop a running (or ready to run) task from another task. Used in the speculative selection pattern discussed in Section 3.6.3.

category reduction pattern: A combination of search and segmented reduction, this is the form of reduction used in the map–reduce programming model. Each input has a label, and reduction occurs only between elements with the same label. The output is a set of reduction results for each unique label. See Section 3.6.8.

circuit complexity: See ***span complexity***.

closures: Objects that consist of a function definition and a copy of the environment (that is, the values of all variables referenced by the function) in effect at the time and visible from the scope in which the function was defined. See ***lambda function*** and Section 3.4.4.

cloud: A set of computers, typically maintained in a data center, that can be allocated dynamically and accessed remotely. Unlike a ***cluster***, cloud computers are typically managed by a third party and may host multiple applications from different, unrelated users.

cluster: A set of computers with distributed memory communicating over a high-speed interconnect. The individual computers are often called ***nodes***.

codec: An abbreviation for coder–decoder, a module that implements a data compression and decompression algorithm in order to reduce memory storage or communication bandwidth. For example, codecs that compress to/from MPEG4 are common for video.

code fusion: An optimization for a sequence of ***vector operations*** that combines the operations into a single ***elemental function***.

coherent masks: When the SPMD programming model is emulated on SIMD machines using ***masking***, the situation where the masks contain all 0's or all 1's.

collective operation: An operation, such as a ***reduction*** or a ***scan***, that acts on a collection of data as a whole. See Chapter 5.

collision: In the ***scatter pattern***, or when using random writes from parallel tasks, a collision occurs when two items try to write to the same location. The result is typically ***non-deterministic*** since it depends on the timing of the writes. In the worst case, a collision results in garbage being written to the location if the writes are not ***atomic*** and are not protected with ***locks***. See Sections 3.5.5 and 6.2.

combiner operation: The (ideally) associative and (possibly) commutative operation used in the definition of ***collective operations*** such as ***reduction*** and ***scan***. See Chapter 5.

communication: Any exchange of data or ***synchronization*** between software tasks or threads. Understanding that communication costs are often a limiting factor in scaling is a critical concept for parallel programming.

communication avoiding algorithm: An algorithm that avoids communication, even if it results in additional or redundant computation.

commutative operation: A commutative operation $\oplus$ satisfies the equation $a \oplus b = b \oplus a$ for all a and b in its domain. Some techniques for vectorizing reductions require commutativity.

composability: The ability to use two components with each other. Can refer to system features, programming models, or software components. See Section 1.5.4.

concurrent: Logically happening simultaneously. Two tasks that are both logically active at some point in time are considered to be concurrent. Contrast with ***parallel***.

continuation: The state necessary to continue a program from a certain logical location in that program. A well-known example is the statement following a subroutine call, which will be where a program continues after a subroutine finishes (returns). The continuation is more than just the location; it also includes the state of data, variable declarations, and so forth at that point.

continuation passing style: A style of programming in which the ***continuations*** of operations are explicitly created and managed.

control dependency: A ***dependency*** between two tasks where whether or not a task executes depends on the result computed by another task.

convergent memory access: When memory accesses in adjacent ***SIMD lanes*** access adjacent memory locations.

cooperative scheduling: A thread scheduling system that allows thread to switch tasks only at predictable switch points.

core: A separate subprocessor on a multicore processor. A core should be able to support (at least one) separate and divergent flow of control from other cores on the same processor. Note that there is some inconsistency in the use of this term. For example, some graphic processor vendors use the term as well for SIMD ***lanes*** supporting ***fibers***. However, the separate flows of control in fibers are simulated with masking on these devices, so there is a performance penalty for divergence. We will restrict the use of the term ***core*** to the case where control flow divergence can be done without penalty.

critical path: The longest chain of ***tasks*** ordered by ***dependencies*** in a program.

DAG: See ***directed acyclic graph***.

data dependency: A ***dependency*** between two tasks where one task requires as input data the output of another task.

data locality: See ***locality.***

data parallelism: An attempt to an approach to parallelism that is more oriented around data rather than tasks. However, in reality, successful strategies in parallel algorithm development tend to focus on exploiting the parallelism in data, because data decomposition (generating tasks for different units of data) scales, but functional decomposition (generation of hetereogeneous tasks for different functions) does not. See ***Amdahl's Law***, ***Gustafson–Barsis' Law***, and Section 2.2.

deadlock: A programming error that occurs when at least two tasks wait for each other and each will not resume until the other task proceeds. This happens easily when code requires locking multiple mutexes; for example, each task can be holding a mutex required by the other task. See Section 2.6.3.

dependencies: A relationship among tasks that results in an ordering constraint. See ***data dependency*** and ***control dependency***.

depth: See ***span complexity***.

deque: A double-ended queue.

design pattern: A general term for ***pattern*** that includes not only ***algorithmic strategy patterns*** but also patterns related to overall code organization.

deterministic: A deterministic algorithm is an algorithm that behaves predictably. Given a particular input, a deterministic algorithm will always produce the same output. The definition of what is the "same" may be important due to limited precision in mathematical operations and the likelihood that optimizations including ***parallelization*** will rearrange the order of operations. These are often referred to as "rounding" differences, which result when the order of mathematical operations to compute answers differs between the original program and the final concurrent program. Concurrency is not the only factor that can lead to ***non-deterministic*** algorithms but in practice it is often the cause. Use of programming models with sequential semantics and eliminating data races with proper access controls will generally eliminate non-determinism other than the "rounding" differences.

directed acyclic graph: A graph that defines a partial order so that nodes can be sorted into a linear sequence with references only going in one direction. A directed acyclic graph has, as its name suggests, directed edges and no cycles.

direct memory access (DMA): The ability of one processing unit to access another processing unit's memory without the involvement of the other processing unit.

direct-mapped cache: A cache in which every location in memory can be stored in only one location in the cache, typically using a modular function of the address.

distributed memory: Memory which is located in multiple physical locations. Accessing data from more remote locations typically has higher ***latency*** and possibly lower ***bandwidth*** than accessing local memory.

distributed memory: Memory that is physically located in separate computers. An indirect interface, such as message passing, is required to access memory on remote computers, while local memory can be accessed directly. Distributed memory is typically supported by ***clusters***, which, for purposes of this definition, we are considering to be a collection of computers. Since the memory on ***attached co-processors*** also cannot typically be addressed directly from the host, it can be considered, for functional purposes, to be a form of distributed memory.

divergent memory access: When memory accesses in adjacent SIMD lanes access non-adjacent memory locations. See ***convergent memory access***.

divide-and-conquer pattern: Recursive decomposition of a problem. Can often be parallelized with the ***fork–join*** parallel pattern. See Section 8.1.

domain-specific language (DSL): A language with specialized features suitable for a specific application domain, along with (typically) some restrictions to make optimization for that domain easier. For instance, an image processing language might support direct specification of the ***stencil pattern*** but restrict the use of general pointers. Domain-specific languages are often ***embedded languages***, in which case they are called *embedded domain-specific languages*, or EDSLs.

dwarf: A workload is which typical of some class of workloads. Sometimes used as a synonym for ***pattern***.

efficiency: Efficiency measures the return on investment in using additional hardware to operate in parallel. See Section 2.5.2.

elemental function: A function used in a ***map pattern***. An elemental function syntactically is defined as acting on single item inputs, but in fact is applied in parallel to all the elements of a collection. An elemental function can be vectorized by replicating the computation it specifies across multiple ***SIMD*** lanes. See Sections 4.1 and B.10.

embarrassing parallelism: Refers to an algorithm that can be decomposed into a large number of independent tasks with little or no synchronization or communication required. See ***map pattern***.

embedded language: A programming system whose syntax is supported using another language; for example, ArBB supports an embedded interface in C++. The computations specified using this interface are not, however, performed by C++. Instead, ArBB supports a set of types and operations in C++. Sequences of these operations can be recorded by ArBB and are then dynamically recompiled to machine language. See Section B.10.

expand pattern: A pattern in which each element of a ***map pattern*** can output zero or more data elements, which are then assembled into a single (possibly segmented) array. Related to the ***pack pattern***. See Sections 3.6.7 and 6.4.

false sharing: Two separate tasks in two separate cores may write to separate locations in memory, but if those memory locations happened to be allocated in the same cache line, the cache coherence hardware will attempt to keep the cache lines coherent, resulting in extra interprocessor communication and reduced performance, even though the tasks are not actually sharing data. See Section 2.4.2.

fiber: A very lightweight unit of parallelism that (conceptually) has its own flow of control but is mapped onto a single lane of a SIMD processor. Divergent control flow between fibers on a single SIMD processor is simulated by masking updates to registers and memory. See Section 2.3. A masked implementation has implications for performance. In particular, divergent control flow reduces lane utilization. There may also be limitations on control flow; for example, GOTO may not be supported, only nested control flow. Note that the term *fiber* is not universally accepted. In particular, on GPUs, fibers are often called ***threads*** and what we call threads are called *work groups* in OpenCL.

Fibonacci numbers: The Fibonacci numbers are defined by linear recurrence relationship and suffer from overuse in computer science as examples of recursion as a result. Fibonacci numbers are defined as $F(0) = 0$ and $F(1) = 1$ plus the relationship defined by $F(N) = F(N-1) + F(N-2)$.

fine-grain locking: Locks that are used to protect parts of a larger data structure from race conditions. Such locks avoid locking the entirety of a large data structure during parallel accesses.

fine scale: A level of parallelism with very small units of parallel work. Reduction of overhead is very important for fine-scale parallelism to be effective, since otherwise the overhead will dominate the computation.

Flynn's characterization: A classic categorization of parallel processors by Flynn [Fly72] based on whether they have multiple flows of control or multiple streams of data. See Section 2.4.3.

fold: A collective operation in which every output is a function of all previous outputs and all inputs up to the current output point. A fold is based on a ***successor function*** that computes a new output value and a new state for the fold for each new input value. A ***scan*** is a special, parallelizable case of a fold where the successor function is associative.

fork: The creation of a new thread or task. The original thread or task continues in parallel with the forked thread or task. See ***spawn***.

fork–join pattern: A pattern of computation in which new (potential) parallel flows of control are created/split with ***forks*** and terminated/merged with ***joins***. See Sections 3.3.1 and 8.1.

fork point: A point in the code where a ***fork*** takes place.

fully associative cache: See ***associative cache***.

functional decomposition: An approach to parallelization of existing serial code where modules are run on different threads. This approach does not give more than a constant factor of speedup at best since the number of modules in a program is fixed.

functional unit: A hardware processing element that can do a simple operation, such as a single arithmetic operation.

functor: A class which supports a function-call interface. Unlike functions in C and C++ however, functors can also hold state and can support additional interfaces to modify that state. See ***lambda functions***.

fusion: An optimization in which two or more things with similar forms are combined. See ***loop fusion, cache fusion***, and ***code fusion***.

future: An approach to asynchronous computing in which a computation is specified but does not necessarily begin immediately. Instead, construction of a future returns an object which can be used to query the status of the computation or wait for its completion.

future-proofed: A computer program written in a manner so it will survive future computer architecture changes without significant changes to the program itself being necessary. Generally, the more abstract a programming method is, the more future-proof that program is. Lower-level programming methods that in some way mirror computer architectural details will be less able to survive the future without change. Writing in an abstract, more future-proof fashion may involve tradeoffs in efficiency, however.

gather pattern: A set of parallel random reads from memory. A gather takes a collection of addresses and an input collection and returns a collection of data drawn from the input collection at the given locations. Gathers are equivalent to random reads inside a ***map pattern***. See Sections 3.5.4 and 6.1.

geometric decomposition pattern: A pattern that decomposes the computational domain for an algorithm into a set of possibly overlapping subdomains. A special case is the ***partition pattern***, which is when the subdomains do not overlap. See Sections 3.5.3 and 6.6.

GPU: A graphics processing unit is an attached graphics processor originally specialized for graphics computations. GPUs are able to support arbitrary computation, but they are specialized for massively parallel, fine-grained computations. They typically use multithreading, multiple threads per core, and **fibers** and make extensive use of ***latency hiding***. They are typically able to maintain the state for many more threads in memory than CPUs, but each thread can have less total state.

grain: A unit of work to be run serially. See ***granularity***.

grain size: The amount of work in a ***grain***.

granularity: The amount of decomposition applied to the ***parallelization*** of an algorithm, and the **grain size** of that decomposition. If the granularity is too coarse, there are not enough parallel tasks to effectively make use of all parallel hardware units and hide latency. If the granularity is too fine, there are too many parallel tasks and overhead may dominate the computation.

graphics accelerators: A processor specialized for graphics workloads, usually in support of real-time graphics APIs such as Direct3D and OpenGL. See ***GPU***.

graph rewriting: A computational pattern where nodes of a graph are matched against templates and substitutions made with other subgraphs. When applied to directed acyclic graphs (trees with sharing), this is known as *term graph rewriting* and is equivalent to the lambda calculus, except that it also explicitly represents sharing of memory. See Section 3.6.9.

greedy scheduling: A scheduling strategy in which no worker idles if there is work to be done.

grid: A distributed set of computers that can be allocated dynamically and accessed remotely. A grid is distinguished from a cloud in that a grid may be supported by multiple organizations and is usually more heterogeneous and physically distributed.

Gustafson–Barsis' Law: A different view on Amdahl's Law that factors in the fact that as problem sizes grow the serial portion of computations tend to shrink as a percentage of the total work to be done. Compare with other attempts to characterize the bounds of parallelism, such as ***Amdahl's Law*** and ***span complexity***. See Section 2.5.5.

halo: In the implementation of the ***stencil pattern*** on ***distributed memory*** a set of elements surrounding a ***partition*** that are replicated on different workers to allow each portion of the partition to be computed in parallel.

hardware thread: A hardware implementation of a task with a separate flow of control. Multiple hardware threads can be implemented using multiple cores, or they can run concurrently or simultaneously on one core in order to hide latency using methods such as ***hyperthreading*** of a processor core. See Sections 1.2 and 2.5.9.

heap allocation: An allocation mechanism that supports unstructured memory allocations of different sizes and at arbitrary times in the execution of a program. Compare with ***stack allocation***.

heterogeneous computer: A computer which supports multiple processors each with specialized capabilities or performance characteristics.

holders: A form of /hyperobject/ useful for managing temporary task-local storage.

host processor: The main control processor in a system, as opposed to any graphics processors or co-processors. The host processor is responsible for booting and running the operating system.

hyperobjects: A mechanism in Cilk Plus to support operations such as reduction that combine multiple objects. See Section B.7. For examples using hyperobjects, see Sections 5.3.5, 8.10, and 11.2.1.

hyperthreading: Multithreading on a single processor core. With hyperthreading, also called simultaneous multithreading, multiple ***hardware threads*** may run on one core and share resources, but some benefit is still obtained from parallelism or concurrency. For example, the processor may draw instructions from multiple hyperthreads to fill superscalar instruction slots, or the processor may switch between multiple hyperthreads in order to hide memory access latency. Typically each hyperthread has, at least, its own register file and program counter, so that switching between hyperthreads is relatively lightweight.

implementation pattern: A pattern that is specific to efficient implementation (usually of some other pattern) using specific hardware mechanisms.

instance: In a ***map pattern*** one invocation of an elemental function on one element of the map.

instruction-level parallelism (ILP) wall: The limits to automatic parallelism given by the amount of parallelism naturally available at the instruction level in serial programs.

intrinsics: Intrinsics appear to be functions in a language but are supported by the compiler directly. In the case of SSE or vector intrinsics, the intrinsic function may map directly to a small number, often one, of machine instructions which the compiler inserts without the overhead of a real function call. For a discussion of SSE intrinsics, see Section 5.3.3.

irregular parallelism: parallelism with dissimiliar tasks with unpredictable dependencies.

iteration pattern: A serial pattern in which the same sequence of instructions is executed repeatedly and in sequence.

join: When multiple flows of control meet and a single flow continues onwards. Not to be confused with a ***barrier***, in which all the incoming flows continue onwards.

join point: A point in the code where a ***join*** takes place.

kernel: A general term for a small section of code that (1) executes a large amount of computation relative to other parts of the program (also known as a hotspot), and/or (2) is the key code sequence for an algorithm.

lambda expression: an expression that returns a ***lambda function***.

lambda function: A lambda function, for programmers, is an anonymous function. Long a staple of languages such as LISP, it was only recently supported for C++ per the C++11 standard. A lambda function enables a fragment of code to be passed to another function without having to write a separate named function or functor. This ability is particularly handy for using TBB. See Section D.2.

lane: An element of a SIMD register file and associated functional unit, considered as a unit of hardware for performing parallel computation. SIMD instructions execute computations across multiple lanes.

latency: The time it takes to complete a task—that is, the time between when the task begins and when it ends. Latency has units of time. The scale can be anywhere from nanoseconds to days. Lower latency is better in general. See Section 2.5.1.

latency hiding: Schedules computations on a processing element while other tasks using that core are waiting for long-latency operations to complete, such as memory or disk transfers. The latency is not actually hidden, since each task still takes the same time to complete, but more tasks can be completed in a given time since resources are shared more efficiently, so throughput is improved. See Section 2.5.9.

latent parallelism: See ***potential parallelism***.

linear speedup: Speedup in which the performance improves directly proportional to the physical processing resources available. Considered to be optimal.

Little's formula: A formula relating parallelism, concurrency, and latency.

livelock: A situation in which multiple workers are active, but are not doing useful work and are not making forward progress. See ***deadlock***.

load balancing: Distributing work across resources so that no resource idles while there is work to be done.

load imbalance: A situation where uneven sizes of tasks assigned to workers results in some workers finishing early and then idling while waiting for other workers to complete larger tasks. See ***load balancing***.

locality: Refers to either ***spatial locality*** or ***temporal locality***. Maintaining a high degree of locality of reference is a key to scaling. See Section 2.6.5.

lock: A mechanism for implementing ***mutual exclusion***. Before entering a mutual exclusion region, a thread must first try to acquire a lock on that region. If the lock has already been acquired by another thread, the current thread must ***block***, which it may do by either suspending operation or spinning. When the lock is released, then the current thread is free to acquire it. Locks can be implemented using ***atomic operations***, which are themselves a form of mutual exclusion on basic operations, implemented in hardware. See Section 2.6.2.

loop-carried dependencies: A dependency that exists between multiple iterations of an ***iteration pattern***.

loop fusion: An optimization where two loops with the compatible indexing executed in sequence can be combined into a single loop.

mandatory concurrency: See ***mandatory parallelism***.

mandatory parallelism: Parallelism that is semantically required for program correctness. See Section 9.6.

many-core processor: A ***multicore*** processor with so many cores that in practice we do not enumerate them; there are just "lots." The term has been generally used with processors with 32 or more cores, but there is no precise definition.

map pattern: Replicates a function that is applied to all elements of a collection, producing a new collection with the same shape as the input. The function being replicated is called an ***elemental function*** since it applies to the elements of an actual collection of input data. See Sections 3.3.2 and Chapter 4.

masking: A technique for emulating ***SPMD*** control flow on ***SIMD*** machines in which elements that are not active are prohibited from updating externally visible state.

megahertz era: A historical period of time during which processors doubled clock rates at a rate similar to the doubling of transistors in a design, roughly every 2 years. Such rapid rise in processor clock speeds ceased at just under 4 GHz (4,000 megahertz) in 2004. Designs shifted toward adding more cores, marking the shift to the ***multicore era***.

member function: A function associated with an object and which can access instance-specific object state.

memory fences: A synchronization mechanism which can ensure that memory operations before the fence are completed and are visible before memory operations after the fence.

memory hierarchy: See ***memory subsystem***.

memory subsystem: The portion of a computer system responsible for moving code and data between the main system memory and the computational units. The memory subsystem may include additional connections to I/O devices including graphics cards, disk drives, and network interfaces. A modern memory subsystem will generally have many levels, including some levels of caching both on and off the processor die. Coherent memory subsystems, which are used in most computers, provide for a single view of the contents of the main system memory despite temporary copies in caches and concurrency in the system. See Section 2.4.1.

memory wall: A limit to parallel scalability given by the fact that memory (and more generally, communication) ***bandwidth*** and in particular ***latency*** are not scaling at the same rate as computation.

merge scatter pattern: In a merge scatter, results that collide while implementing a ***scatter pattern*** are combined with an associative operator. The operator needs to be associative so the answer is the same regardless of the order in which elements are combined. We might also want to use this operator to combine scattered values with the previous contents of the target array. The merge scatter pattern can be used to implement histograms, for example. See Section 6.2.

metaprogramming: The use of one program to generate or manipulate another, or itself. See also ***template metaprogramming***.

method: See ***member function***.

MIC: The Intel Many Integrated Core architecture is designed for highly parallel workloads. The architecture emphasizes higher core counts on a single die, and simpler more efficient cores, than on a traditional CPU. A prototype with up to 32 cores and based on 45-nm process technology, known as Knight Ferry, was made available, but not sold, by Intel in 2010 and 2011. A product built on 22-nm process technology with more than 50 cores is expected in late 2012 or sometime in 2013.

MIMD: Multiple Instruction, Multiple Data, one of Flynn's classes of computer that supports multiple threads of control, each with its own data access. See ***SIMD*** and Section 2.4.3.

monoid: An ***associative operation*** that has an identity.

Moore's Law: Describes a long-term trend that the number of transistors that can be incorporated inexpensively on an integrated circuit chip doubles approximately every 2 years. It is named for Intel co-founder Gordon Moore, who described the trend in his 1965 paper in *Electronics Magazine*. This forecast of the pace of silicon technology has essentially described the basic business model for the semiconductor industry as well as being a driving force of technological and social change since the late 20th century.

motif: Sometimes used as a synonym for ***pattern***.

multicore: A processor with multiple subprocessors, each subprocessor (known as a ***core***) supporting at least one hardware thread.

multicore era: Time after which processor designs shifted away from rapidly rising clock rates and shifted toward adding more cores. This era began roughly in 2005.

multiple-processor systems: A system with two or more processors implemented on separate physical dies.

mutex: Short for ***/mutual exclusion/***, and also used as a synonym for ***lock***.

mutual exclusion: A mechanism for protecting a set of data values so that while they are manipulated by one parallel thread they cannot be manipulated by another. See ***lock*** and ***transactional memory***.

nesting pattern: Refers to the ability to hierarchically compose other patterns. The nesting pattern simply means that all "tasks" in the pattern diagrams within this book are actually locations within which general code can be inserted. This code can in turn be composed of other patterns.

Network interface controller (NIC): A specialized communication processor.

node (in a cluster): A shared memory computer, often on a single board with multiple processors, that is connected with other nodes to form a ***cluster*** computer or supercomputer.

non-deterministic: Exhibiting a lack of deterministic behavior, so results can vary from run to run of an algorithm. See more in the definition for ***deterministic***.

non-uniform memory access (NUMA): A memory system in which certain banks of memory take longer to access than others, even though all the memory uses a single address space. See also ***distributed memory***.

objects: Objects are a language construct that associate data with the code to act on and manage that data. Multiple functions may be associated with an object and these functions are called the ***methods*** or ***member functions*** of that object. Objects are considered to be members of a class of objects, and classes can be arranged in a hierarchy in which subclasses inherit and extend the features of superclasses. The state of an object may or may not be directly accessible; in many cases, access to an object's state may only be permitted through its methods. See Section 3.4.5.

offload: Placing part of a computation on an ***attached device*** such as a GPU or co-processor.

online: An algorithm which can begin execution before all input data is read.

OpenCL: Open Computing Language, initiated by Apple Corporation, is now a standard defined by the Khronos group for graphics processors and ***attached co-processors***. However, OpenCL can also be used to specify parallel and vectorized computations on multicore host processors.

optional parallelism: Parallelism that is specified by a programming model but is not semantically necessary. Antonym is ***mandatory parallelism***.

over-decomposition: A parallel programming style where many more tasks are specified than there are physical workers for executing it. This can be beneficial for ***load balancing*** particularly in systems that support ***optional parallelism***.

over-subscription: More threads run on a system than it has physical workers, resulting in excessive overhead for switching between multiple threads or exceeding the number of threads that can be supported by the operating system. This can be avoided by using a programming model with ***optional parallelism***.

pack pattern: A data management pattern where certain elements of a collection are discarded and the remaining elements are placed in a contiguous sequence, maintaining the order of the original input. Related to the ***expand pattern***.

page: The granularity at which virtual to physical address mapping is done. Within a page, the mapping of virtual to physical memory addresses is continuous. See Section 2.4.1.

parallel: Physically happening simultaneously. Two tasks that are both actually doing work at some point in time are considered to be operating in parallel. When a distinction is made between ***concurrent*** and ***parallel***, the key is whether work can ever be done simultaneously. Multiplexing of a single processor core, by multitasking operating systems, has allowed concurrency for decades even when simultaneous execution was impossible because there was only one processing core.

parallel pattern: ***Patterns*** arising specifically in the specification of parallel applications. Examples of parallel patterns include the ***map pattern***, the ***reduction pattern***, the ***fork–join pattern***, and the ***partition pattern***.

parallel slack: The amount of "extra" parallelism available above the minimum necessary to use the parallel hardware resources. See Sections 2.4.2 and 2.5.6.

parallelism: Doing more than one thing at a time. Attempts to classify types of parallelism are numerous; read more about classifications of parallelism in Sections 2.2 and 2.3.

parallelization: The act of transforming code to enable simultaneous activities. The parallelization of a program allows at least parts of it to execute in parallel.

partition pattern: A pattern that decomposes the computational domain for an algorithm into a set of non-overlapping subdomains called ***tiles*** or ***blocks*** (although *tile* is the term preferred in this book). See the ***geometric decomposition pattern***, which is similar but allows overlap between subdomains. The partition pattern is a special case of the geometric decomposition pattern that does not allow overlap. See Section 6.6.

pattern: A recurring combination of data and task management, separate from any specific algorithm. Patterns are universal in that they apply to and can be used in any programming system. Patterns have also been called ***dwarfs***, ***motifs***, and algorithmic skeletons. Patterns are not necessarily tied to any particular hardware architecture or programming language or system. Examples of patterns include the ***sequence pattern*** and the ***object pattern***. See ***parallel pattern*** and Chapter 3.

PCIe bus: A peripheral bus supporting relatively high bandwidth and DMA, often used for attaching specialized co-processors such as ***GPUs*** and ***NICs***.

permutation scatter pattern: A form of the ***scatter pattern*** in which multiple writes to a single storage location are illegal. This form of scatter is deterministic, but can only be considered safe if collisions are checked for. See Section 6.2.

pipeline pattern: A set of data processing elements connected in series, generally so that the output of one element is the input of the next one. The elements of a pipeline are often executed concurrently. Describing many algorithms, including many signal processing problems, as pipelines is generally quite natural and lends itself to parallel execution. However, in order to scale beyond the number of pipeline stages, it is necessary to exploit parallelism within a single pipeline stage. See Sections 3.5.2, 9.2, 12.2, and C.6.

potential parallelism: At a given point of time, the number of parallel tasks that could be used by a parallel implementation of an algorithm, given sufficient hardware resources. Additional hardware resources above the potential parallelism in an algorithm are not usable. If the potential parallelism is larger than the physical parallelism, then the tasks will need to share physical resources by ***serialization***. Also known as ***latent parallelism*** and ***available parallelism***.

power wall: A limit to the practical clock rate of serial processors given by thermal dissipation and the non-linear relationship between power and switching speed.

pragma: A form of program markup used to give a hint to a compiler but not change the semantics of a program. Also called a "compiler directive."

precision: The detail in which a quantity is expressed. Lack of precision is the source of rounding errors in computation. The finite number of bits used to store a number requires some approximation of the true value. Errors accumulate when multiple computations are made to the data in operations such as reductions. Precision is measured in terms of the number of digits that contain meaningful data, known as significant digits. Since precision is most often considered in reference to floating-point numbers, significant digits in computer science have often been measured in bits (binary digits) because most floating-point arithmetic is done in radix-2. With the advent of IEEE-754-2008, radix-10 arithmetic is once again popular and precision of such data would be expressed in terms of decimal digits. See Section 5.1.4.

preemptive scheduling: A scheduling system that allows a thread to switch tasks at any time.

priority scatter pattern: A deterministic form of the ***scatter pattern*** in which an attempt to write multiple values in parallel to a single storage location results in one value (and only one) value being stored based on a priority function, while all other values are discarded. The unique priority given to each parallel write in a priority scatter can be assigned in such a way that the result is deterministic and equivalent to a serial implementation. See Section 6.2.

process: A application-level unit of parallel work. A process has its own thread of control and is managed by the operating system. Usually, unless special provisions are made for ***shared memory***, a process cannot access the memory of another process.

producer–consumer: A relationship in which the producer creates data that is passed to the consumer to utilize or further process. If data is not consumed exactly when it is produced, it must be buffered. Buffering introduces challenges of stalling the producer when the buffer is full, and stalling the consumer when the buffer is empty.

pure function: A function whose output depends only on its input, and that does not modify any other system state.

race condition: Non-deterministic behavior in a parallel program that is generally a programming error. A race condition occurs when concurrent tasks perform operations on the same memory location without proper synchronization and one of the memory operations is a write. Code with a race may operate correctly sometimes and fail other times. See Section 2.6.1.

recurrence pattern: A sequence defined by a recursive equation. In a recursive equation, one or more initial terms are given and each further term of the sequence is defined as a function of the preceding terms. Implementing recurrences with recursion is often inefficient since it tends to recompute elements of the recurrence unnecessarily. Recurrences also occur in loops with dependencies between iterations. In the case of a single loop, if the dependence is associative, it can be ***parallelized*** with the ***scan pattern***. If the dependence is inside a multidimensional loop nest, the entire nest can always be parallelized over $n-1$ dimensions using a hyperplane sweep, and it can also often be parallelized with the fork–join pattern. See Sections 3.3.6, 7.5, and 8.12.

recursion: The act of a function being re-entered while an instance of the function is still active in the same thread of execution. In the simplest and most common case, a function directly calls itself, although recursion can also occur between multiple functions. Recursion is supported by storing the state for the continuations of partially completed functions in dynamically allocated memory, such as on a stack, although if higher-order functions are supported a more complex memory allocation scheme may be required. Bounding the depth of recursion can be important to prevent excessive use of memory.

reduce: Apply operation to merge a collection of values to a single value. An example is summing a sequence of values. See ***reduction pattern***.

reducers: Hyperobjects that can implement ***reduce*** operations.

reduction pattern: The most basic ***collective*** pattern, a reduction combines all the elements in a collection into a single element using pairwise applications of a ***combiner operation***. In order to allow ***parallelization***, the combiner operation should be associative. In order to allow for efficient ***vectorization***, it is useful if the combiner operation is also commutative. Many useful reduction operations, such as maximum and (modular integer) addition, are both associative and commutative. Unfortunately, floating-point addition and multiplication are not, which can lead to potential ***non-determinism***. See Section 5.1.

reduction variable: A variable that appears in a loop for combining the results of many different loop iterations.

refactoring: Reorganizing code to make it better suited for some purpose, such as parallelization.

registers: Very fast but usually very limited on-core storage for intermediate results.

regular parallelism: A class of algorithms in which the tasks and data dependencies are arranged in a regular and predictable pattern.

relative speedup: Speedup in which a parallel solution to a problem is compared to a serialization of the same solution, that is, using the same algorithm. See ***absolute speedup***.

relaxed sequential semantics: See ***sequential semantics*** for an explanation.

response time: The time between when a request is made and when a response is received.

rotate pattern: A special case of the ***shift pattern*** that handles boundary conditions by moving data from the other side of the collection. See Section 6.1.2.

safety: A system property that automatically guards against certain classes of programmer errors, such as race conditions.

saturation: Saturation arithmetic has maximum and minimum values that are utilized when computation would logically arrive at higher or lower values if unbounded numerical representations were utilized. Saturation arithmetic is needed only because numerical representations on computer systems are almost always limited in precision and range. In floating-point arithmetic, the concept of positive and negative infinity as uniquely represented numbers in the floating-point format is utilized and is the default in instances of saturation. In integer arithmetic, wrap-around arithmetic is generally the default. Special instructions for saturation arithmetic are available in modern instruction sets (such as MMX), often originally motivated by graphics where the desire to make a graphical pixel brighter and brighter by increasing the value of a pixel was frustrated by a sudden dimming of the pixel due to wrap-around arithmetic. In an 8-bit unsigned number format, the addition of 254 with 9 will result in an answer of 7 in wrap-around or 255 in saturation arithmetic. Likewise, the subtraction of 11 from 7 would result in 252 in wrap-around vs. 0 in saturation arithmetic. Note, however, that saturation arithmetic for signed numbers is not associative.

scalability: A measure of the increase in performance as a function of the availability of more hardware to use in parallel. See Section 2.5.2.

scalable: An application is ***scalable*** if its performance increases when additional parallel hardware resources are added. See ***scalability***.

scalar promotion: When a scalar and a vector are combined using a vector operation, the scalar is automatically treated as a vector with all elements set to the same value.

scan pattern: Pattern arising from a one-dimensional recurrence relationship in the definition of a computation. This often arises as a loop-carried dependency where the computation of one iteration is dependent on the results of a prior iteration. Such loops are, surprisingly, still parallelizable if the dependency can be expressed as an associative operation. See Section 5.4.

scatter pattern: A set of input data and a set of indices is given, and each element of the input is written at the given location. Scatter can be considered the inverse of the ***gather pattern***. A collision in output occurs if the set of indices maps multiple input data to the same location. There are at least four ways to resolve such collisions: ***permutation scatter***, ***atomic scatter***, ***priority scatter***, and ***merge scatter***. See Section 3.5.5.

search pattern: A pattern that finds data that meets some criteria within a collection of data. See Section 3.6.5.

segmentation: A representation of a collection divided into non-uniform non-overlapping subdomains. Operations such as reduction and scan can be generalized to operate over the segments of a collection independently while still being perfectly load balanced. See Section 3.6.6.

selection pattern: A serial pattern in which one of two flows of control are chosen based on a Boolean control expression.

semantics: What a programming language construct does, as opposed to how it does it (pragmatics) or how it is expressed (syntax).

separating hyperplane: A plane that can be used to determine the sweep order for executing a multidimensional ***recurrence*** in parallel.

sequence pattern: The most fundamental serial pattern in which tasks are executed one after the other, with each task completing before the next one starts. See Section 3.2.1.

sequential bottlenecks: See ***serial bottlenecks***.

sequential consistency: Sequential consistency is a memory consistency model where every task in a concurrent system sees all memory writes (updates) happen in the exact same order, and a task's own writes occur in the order that the task specified. See Section 2.6.1.

sequential semantics: Refers to when a (parallel) program can be executed using a single thread of control as an ordinary sequential program without changing the semantics of the program. Parallel programming with sequential semantics has many advantages over programming in a manner that precludes serial execution and is therefore strongly encouraged. Such programs are considered easier to understand, easier to debug, more efficient on sequential machines, and better at supporting nested parallelism. Sequential semantics casts parallelism as an accelerator and not as mandatory for correctness. This means that one does not need a conceptual parallel model to understand or execute a program with sequential semantics. Examples of ***mandatory parallelism*** include producer–consumer relationships with bounded buffers (hence, the producer cannot necessarily be completely executed before the consumer because the producer can become blocked) and message passing (e.g., MPI) programs with cyclic message passing. Due to timing, precision, and other sources of inexactness, the results of a sequential execution may differ from the concurrent invocation of the same program. Sequential semantics solely means that any such variation is not due to the semantics of the program. The term "relaxed sequential semantics" is sometimes used to explicitly acknowledge the variations possible due to non-semantic differences in serial vs. concurrent executions. See Section 1.1 See ***serial semantics.***

serial: Neither concurrent nor parallel.

serial bottlenecks: A region of an otherwise parallel program that runs serially.

serial consistency: A parallel program that produces the same result as a specific serial ordering of its tasks.

serial elision: The serial elision of a Cilk Plus program is generated by erasing occurrences of the `cilk_spawn` and `cilk_sync` keywords and replacing `cilk_for` with `for`. Cilk Plus is a faithful extension of C/C++ in the sense that the serial elision of any Cilk Plus program is both a serial C/C++ program *and* a semantically valid implementation of the Cilk Plus program. The term *elision* arose from earlier versions of Cilk that lacked `cilk_for`, so eliding (omitting) the two other keywords sufficed. The term "C elision" is sometimes used, too, harking back to when Cilk was an extension of C but not C++. See Section B.4.

serial illusion: The apparent serial execution order of machine language instructions in a computer. In fact, hardware is naturally parallel, and many low-level optimizations and high-performance implementation techniques can reorder operations.

serial semantics: Same as ***sequential semantics***.

serial traps: A serial trap is a programming construct that semantically requires serial execution for proper results in general even though common cases may be overconstrained with regard to concurrency by such semantics. The term "trap" acknowledges how such constructs can easily escape attention as barriers to parallelism, in part because they are so common and were not intentionally designed to preclude parallelism. For instance, `for`, in the C language, has semantics that dictate the order of iterations by allowing an iteration to assume that all prior iterations have been executed. Many loops do not rely upon side-effects of prior iterations and would be natural candidates

for parallel execution, but they require analysis in order for a system to determine that parallel execution would not violate the program semantics. Use of `cilk_for`, for instance, has no such serial semantics and therefore is not a serial trap. See examples in Section 1.3.3.

serialization: Refers to when the tasks in a potentially parallel algorithm are executed in a specific serial order, typically due to resource constraints. The opposite of ***parallelization.***

set associative cache: A cache architecture in which a particular location in main memory can be stored in a (small) number of different locations in cache.

shared address space: Even if units of parallel work do not share a physical memory, they may agree on conventions that allow a single unified set of addresses to be used. For example, one range of addresses could refer to memory on the host, while another range could refer to memory on a specific co-processor. The use of unified addresses simplifies memory management.

shared memory: Refers to when two units of parallel work can access data in the same location. Normally doing this safely requires ***synchronization***. The units of parallel work—***processes***, ***threads***, ***tasks***, and ***fibers***—can all share data this way, if the physical memory system allows it. However, processes do not share memory by default and special calls to the operating system are required to set it up.

shift pattern: A special case of the ***gather pattern*** that translates (that is, offsets the location of) data in a collection. There are a few variants based on how boundary conditions are handled. The basic pattern fills in a default value at boundaries, while the ***rotate pattern*** moves data from the other side of the collection. See Section 6.1.2.

SIMD: Single Instruction, Multiple Data, one of Flynn's classes of computer that supports a single operation over multiple data elements. See ***MIMD*** and Section 2.4.3.

simultaneous multithreading: A technique that supports the execution of multiple threads on a single core by drawing instructions from multiple threads and scheduling them in each superscalar instruction slot.

SIMT: Single Instruction, Multiple Threads, a variation on Flynn's characterizations that is really a collection of multiple SIMD processors, with control flow emulated on SIMD machines using a mechanism such as masking. See Section 2.4.3.

software thread: A software thread is a virtual hardware thread—in other words, a single flow of execution in software intended to map one for one to a hardware thread. An operating system typically enables many more software threads to exist than there are actual hardware threads by mapping software threads to hardware threads as necessary. See Section 2.3.

space complexity: A complexity measure for the amount of memory used by an algorithm as a function of problem size.

span: How long a program would take to execute on an idealized machine with an infinite number of processors. The ***span*** of an algorithm can also be seen as the critical path in its task dependency graph. See ***span complexity***.

span complexity: Span complexity is an asymptotic measure of complexity based on the ***span***. In the analysis of parallel algorithms and in particular in order to predict their ***scalability***, this measure is as important as ***work complexity***. Other synonyms for ***span complexity*** in the literature are ***step complexity***, ***depth***, or ***circuit complexity***. Compare with other attempts to characterize the bounds of parallelism: ***Amdahl's Law*** and ***Gustafson-Barsis' Law***. See Section 2.5.6.

spatial locality: Nearby when measured in terms of distance (in memory address). Compare with ***temporal locality***. Spatial locality refers to a program behavior where the use of one data element

indicates that data nearby, often the next data element, will probably be used soon. Algorithms exhibiting good spatial locality in data usage can benefit from cache line structures and prefetching hardware, both common components in modern computers.

spawn: Generically, the creation of a new ***task***. In terms of Cilk Plus, `cilk_spawn` creates a spawn, but the new task created is actually the ***continuation*** and not the call that is the target of the spawn keyword. See ***fork***.

spawning block: The function, try block, or `cilk_for` body that contains the spawn. A sync (`cilk_sync`) waits only for spawns that have occurred in the same spawning block and have no effect on spawns done by other tasks or threads, nor those done prior to entering the current spawning block. A sync is always done, if there have been spawns, when exiting the enclosing spawning block.

speedup: Speedup is the ratio between the latency for solving a problem with one processing unit versus the latency for solving the same problem with multiple processing units in parallel. See Section 2.5.2.

split pattern: A generalized version of the ***pack pattern*** that takes an input collection and a set of Boolean labels to go with every element of that collection. It reorganizes the data so all the elements marked with `false` are at one end of the output collection (rather than discarding them as with the pack pattern), and all the elements marked with `true` are at the other end of the collection. The determinisitic version of this pattern is stable, in that it preserves the original order of the input collection in each output partition. One major application of this pattern is in base-2 radix sort. The ***bin pattern*** is a generalization to more than two categories. See Section 6.4.

SPMD (Single Program, Multiple Data): A programming system that runs a single function on multiple programming elements, but allows each instance of the function to follow different control flow paths. See also ***SIMD, MIMD,*** and ***SIMT***.

stencil pattern: A regular input data access pattern based on a set of fixed offsets relative to an output position. The stencil is repeated for every output position in a grid. This pattern combines the ***map pattern*** with a local ***gather*** over a fixed set of relative offsets and can optionally be implemented using the ***shift pattern***. Stencil operations are common in algorithms that deal with regular grids of data, such as image processing. For example, convolution is an image processing operation where the inputs from a stencil are combined linearly using a weighted sum. See Chapter 7.

step complexity: See ***span complexity***.

strand: In Cilk Plus, a serially executed sequence of instructions that does not contain a spawn or sync point. In the directed acyclic graph model of Section 2.5.2, a strand is a vertex with at most one outgoing and at most one incoming edge. A `cilk_spawn` ends the current strand and starts two new strands, one for the callee and one for the continuation of the caller. A `cilk_sync` ends one or more strands and starts a new strand for the continuation after the join.

strangled scaling: A programming error in which the performance of parallel code is poor due to high contention or overhead, so much so that it may underperform the non-parallel (serial) code. See Section 2.6.4.

strip-mining: When implementing a stencil or map, an optimization that groups instances in a way that avoids unnecessary and redundant memory accesses and aligns memory accesses with vector lanes.

strong scalability: A form of scalability that measures how performance increases when using additional workers but with a fixed problem size. See ***Amdahl's Law*** and ***weak scalability***.

structure-of-arrays (SoA): A data layout for collections of heterogeneous data where all the data for each component of each element of the collection is stored in adjacent physical locations, so that data of the same type is stored together. Compare with ***array-of-structures***.

successor function: In a ***fold***, the function that computes a new state given the old state and a new input item.

superlinear speedup: Speedup where performance grows at a rate greater than the rate at which new workers are added. Since linear scalability is technical optimal, superlinear speedup is typically the result of cache effects, changes in the algorithm behavior, or speculative execution.

superscalar processor: A processor that can execute multiple instructions in a single clock cycle.

superscalar sequence pattern: A sequence of tasks ordered by data dependencies rather than being ordered by a single sequential ordering. This allows parallel (superscalar) execution of tasks that have no relative ordering relative to each other. See Sections 3.6.1.

switch-on-event multithreading: A technique that supports the execution of multiple threads on a single core by switching to another thread on a long-latency event, such as a cache miss.

sync: In terms of Cilk Plus, `cilk_sync` creates a sync point. Control flow pauses at a sync point until completion of all spawns issued by the spawning block that contains the sync point. A sync is not affected by spawns done by other tasks or threads, nor those done prior to entering the current spawning block. An sync is always done when exiting a spawning block that contained any spawns. This is required for program ***composability***.

synchronization: The coordination, of tasks or threads, in order to obtain the desired runtime order. Commonly used to avoid undesired ***race conditions***.

tail recursion: A form of recursion where a result of the recursive call is returned immediately without modification to the parent function. Such uses of recursion can be converted to ***iteration***.

target processor: A (typically specialized) processor to which work can be ***offloaded***. See ***host processor***.

task: A lightweight unit of potential parallelism with its own control flow. Unlike threads, tasks usually do not imply mandatory parallelism. Threads are a mechanism for executing tasks in parallel, whereas tasks are units of work that merely provide the *opportunity* for parallel execution; tasks are not themselves a mechanism of parallel execution.

task parallelism: An attempt to classify parallelism as more oriented around tasks than data. We deliberately avoid use of this term because its meaning varies. In particular, elsewhere "task parallelism" can refer to tasks generated by functional decomposition *or* to irregular tasks still generated by data decomposition. In this book, any parallelism generated by data decomposition, regular or irregular, is considered data parallelism. See Section 2.2.

template metaprogramming: The use of generic programming techniques to manipulate and optimize source code before it is compiled. Specifically, the template rewriting rules in C++ can be interpreted as a functional language for manipulating C++ source code. Some high-performance libraries make use of this fact to automatically perform optimizations of C++ code by, for example, fusing operations together. See the more general term ***metaprogramming***.

temporal locality: Nearby when measured in terms of time; compare with ***spatial locality***. Temporal locality refers to a program behavior in which data is likely to be reused relatively soon. Algorithms exhibiting good temporal locality in data usage can benefit from the data caching common in modern computers. It is not unusual to be able to achieve both temporal and spatial locality in data usage. Computer systems are generally more able to achieve optimal performance when both are achieved, hence the interest in algorithm design to do so.

thread: In general, a *software thread* is any software unit of parallel work with an independent flow of control, and a *hardware thread* is any hardware unit capable of executing a single flow of control (in particular, a hardware unit that maintains a single program counter). Threads are a mechanism for implementing tasks. A multitasking or multithreading operating system will multiplex multiple software threads onto a single hardware thread by interleaving execution via software-created time-slices. A multicore or many-core processor consists of multiple cores to execute at least one independent software thread per core through duplication of hardware. A multithreaded or hyper-threaded processor core will multiplex a single core to execute multiple software threads through interleaving of software threads via hardware mechanisms.

thread parallelism: A mechanism for implementing parallelism in hardware using a separate flow of control for each task. See Section 2.3.

throughput: Given a set of tasks to be performed, the rate at which those tasks are completed. Throughput measures the rate of computation, and it is given in units of tasks per unit time. See ***bandwidth*** and ***latency*** and Section 2.5.1.

tile: A region of memory, typically a section of a larger collection, such as might result from the application of the ***partition pattern***. See ***granularity***, ***block***, and ***tiling***.

tiled decomposition: See ***tiling.***

tiled SIMD: Execution of an ***SPMD*** program using an array of ***SIMD*** processors, each such processor with a separate thread of control.

tiling: Dividing a loop into a set of parallel tasks of a suitable ***granularity***. In general, tiling consists of applying multiple steps on a smaller part of a problem instead of running each step on the whole problem one after the other. The purpose of tiling is to increase the reuse of data in caches. Tiling can lead to dramatic performance increases when a whole problem does not fit in cache. We prefer the term "tiling" to "blocking" and "tile" rather than "block." Tiling and tile have become the more common term in recent times. Sections 5.1.3 and 7.3 for more discussion.

time complexity: A complexity measure for the amount of time used by an algorithm as a function of problem size.

TLB: A Translation Lookaside Buffer is a specialized cache used to hold translations of virtual to physical page addresses. The number of elements in the TLB determines how many pages of memory can be accessed simultaneously with good efficiency. Accessing a page not in the TLB will cause a TLB miss. A TLB miss typically causes a trap to the operating system so that the page table can be referenced and the TLB updated. See Section 2.4.1.

TLB miss: Occurs when a virtual memory access is made for which the page translation is not available in the ***TLB***.

TLB thrashing: The overhead caused by the high ***TLB miss*** rate that results when a program frequently accesses more pages than can be covered by a ***TLB***.

transaction: An atomic update to data, meaning that the results of the update either are not seen or are seen in their entirety. Transactions satisfy the need for atomic data updates to a central repository without requiring an ordering on the updates. Transactions are motivated by the need to have updates be observed in an "all or nothing" fashion. Consider an update to a hotel reservation in an online system, from an "economy room for $75/night" to a "penthouse suite for $9800/night." We do not want a separate task to see a partial update and bill us for $9800/night for an economy room. In general, transaction operations will be non-associative and the outcome will not be deterministic if the order in which the individual operations are performed is non-deterministic. The ***merge***

scatter pattern with a non-associative operator can result in simple forms of the transaction pattern. See Sections 3.7.2 and 6.2.

transactional memory: A way of accessing memory so that a collection of memory updates, called a *transaction*, will be visible to other tasks or threads all at once. Additionally, for a transaction to succeed, any data read during the transaction must not be modified during the transaction by other tasks or threads. Transactions that fail are generally retried until they succeed. Transactional memory offers an alternative method of mutual exclusion from traditional locking that may enhance the scalability of an algorithm in certain cases. Intel Transactional Support Extensions (TSX) support is an example of hardware support for transactional memory.

Translation Lookaside Buffer: See ***TLB***.

uniform parameter: A parameter that is broadcast to all the elements of a map and therefore is the same for each instance of the map's ***elemental function***. See ***varying parameter***.

unpack pattern: The inverse of the ***pack pattern***, this operation scatters data back into its original locations. It may optionally fill in a default value for missing data.

unsplit pattern: The inverse of the ***split pattern***, this operation scatters data back into its original locations. Unlike the case with the ***unpack pattern***, there is no missing data to worry about.

unzip pattern: The inverse of the ***zip pattern***, this operation deinterleaves data and can be used to convert from array-of-structures to structure-of-arrays.

varying parameter: A parameter to a ***map pattern*** that delivers a different element to each instance of the map's ***elemental function***. See ***uniform parameter***.

vector intrinsics: An ***instrinsic*** used to specify a ***vector operation***.

vector operation: A low-level operation that can act on multiple data elements at once in ***SIMD*** fashion.

vector parallelism: A mechanism for implementing parallelism in hardware using the same flow of control on multiple data elements. See Section 2.3.

vector processor: A form of SIMD processor in which large amounts of data are streamed to and from external memory. True vector processors are rare today, so this term now is also used for processors with SIMD instructions that can act on short, fixed-length vectors held in registers.

vectorization: The act of transforming code to enable simultaneous computations using vector hardware. Instructions such as MMX, SSE, and AVX instructions utilize vector hardware. The vectorization of code tends to enhance performance because more data is processed per instruction than would be done otherwise. Vectorization is a specialized form of parallelism. See also ***vectorize***.

vectorize: Converting a program from a scalar implementation to a vectorized implementation to utilize vector hardware such as SIMD instructions (MMX, SSE, AVX, etc.).

vector units: ***functional units*** that can issue multiple operations of the same type in a single clock cycle in ***SIMD*** fashion.

virtual memory: Virtual memory decouples the address used by software from the physical addresses of real memory. The translation from virtual addresses to physical addresses is done in hardware which is initialized and controlled by the operating system. See Section 2.4.1.

VLIW (Very Large Instruction Word): An processor architecture which supports instructions which can explicitly issue multiple operations in a single clock cycle. See ***superscalar processor.***

weak scalability: A form of scalability that measures how performance increases when using additional workers with a problem size that grows at the same rate. See ***Gustafson-Barsis's Law*** and ***strong scalability.***

work: The computational part of a program, as contrasted with communication or coordination. An abstract unit of such computation.

work complexity: The asymptotic number of operations required by an algorithm to run on a single thread. Work complexity is essentially the traditional asymptotic complexity for sequential running time, although frequently, so speedup ratios can be computed, it is better to use ***big Theta notation*** rather than ***big O notation***. Related terms include ***span complexity***.

worker: An abstract unit of actual parallelism, for example, a ***core*** or a ***SIMD lane***.

working set: For an algorithm, the set of data that should be maintained in cache for good performance.

work-span: A model for parallel computation that can be used to compute both upper and lower bounds on speedup. See Section 2.5.6. Related terms include ***span complexity*** and ***work complexity***.

workpile pattern: An extension of the ***map pattern*** that allows new work items to be added during execution from inside the elemental function. If the map pattern can be thought of as a ***parallelization*** of a `for` loop, the workpile pattern can be thought of as a generalization of a `while` loop. See Section 3.6.4.

work-stealing: A ***load balancing*** technique where ***/workers/*** that become idle search for and "steal" pending work from other, busy workers.

zip pattern: A special case of the ***gather pattern*** that interleaves elements from collections, converting from structure-of-arrays to array-of-structures. See Section 6.1.3.