Scientific Computing (Spring 2023)
Parallel Hardware and Parallel Software

1 Von Neumann Architecture and Modifications

1.1 Classical von Neumann Architecture

- Main memory
  - Collection of locations
  - Each capable of storing instructions and data
    - every location consists of an address (used to access the location), and
    - the content of the location (instructions or data stored in the location)
- Central-processing unit (CPU) or processor or core. It is divided into
  - a control unit (responsible for deciding which program instructions should be executed) and
  - an arithmetic and logic unit (or ALU, responsible for executing the actual instructions)

  Data in the CPU and information about the state of an executing program are stored in a very fast storage called registers. The control unit has a special register named program counter, that stores the address of the next instruction to be executed.
- Interconnection (between memory and CPU)
  - Traditionally a bus (collection of parallel wires and some hardware controlling access to the wires) for transferring instructions and data between CPU and memory.

  A von Neumann machine executes a single instruction at a time, and each instruction operates only on a few pieces of data.
  - Transfer from memory to CPU: data or instructions fetched or read from memory.
  - Transfer from CPU to memory: data are written to memory or stored.

  The separation of memory and CPU is called von Neumman bottleneck, since interconnection determines the rates at which instructions and data can be accessed. Data and instructions needed to run a program is effectively isolated from CPU. Difference in speed between executing and fetching.

1.2 Operative System (OS)

Software interface between the user and computer hardware. It determines

- which programs can run and when
- controls the allocation of memory to running programs
- controls access to peripheral devices

When a user runs a program, the OS creates a process (an instance of a computer program that is being executed). A process consists of

- The executable machine language program
- A block of memory (which include the executable code), a call stack that tracks active functions, a heap (to store variables of a running program), and some other memory locations
- Descriptors of resources that OS allocated to the process (e..g., file descriptors)
- Security information (e.g., which hardware and software resources the process can access)
- Information about the state of the process

Multitasking OS:

- OS provides support for the apparent simultaneous execution of multiple programs
- Each process runs for a small time interval (time slice)
- After one process has executed for a time slice, the OS can run a different program
- If a process needs to wait for a resource, it will block (it will stop executing and the OS can run another process)
- However, many programs can continue to do useful work even though the part of the program that is currently executing must wait on a resource
- Threading provides a mechanism to divide programs into more or less independent tasks with the property that when a thread is blocked, another thread can be run
- In general, it is faster to switch between threads than to switch between processes, because they require less resources than processes (lighter weight)
- Threads are contained within processes, so they can use the same executable, and usually share the same memory and I/O devices
- Threads need a record of their own program counters and call stacks to execute independently of each other
- A process can be considered the master thread of execution and threads are started (forks off the process) and stopped or terminated (joins the process)

1.3 Addressing von Neumann bottleneck: Caching

Some improvements are targeted at reducing the von Neumann bottleneck and others at making CPUs faster.

- A cache is a collection of memory locations that can be accessed in less time than some other memory locations
- CPU cache is a collection of memory locations the CPU can access more quickly than main memory
- A CPU cache can be located on the same chip as the CPU or it can be located on a separate chip that can be accessed much faster than an ordinary memory chip

What to store in the cache? locality: After executing an instruction, programs typically execute next instruction (branching is rare) which is on a nearby location (spatial locality) in the near future (temporal locality).

To exploit locality, systems use wider interconnect to access data and instructions (memory access operates on blocks of data and instructions instead of individual data items or instructions)

- Blocks are called cache blocks or cache lines (typically 8/16 times as much as a single memory location)
- Cache is usually divided into levels: the first level (L1) is the smallest and fastest and higher levels (L2, L3, ...) are larger and slower
- Caches usually store copies of information in slower memory (for instance a variable in L1 will also be stored in L2), but there are some multilevel caches that do not duplicate information and in that case the variable in L1 is stored in main memory
- When CPU needs instruction or data it works down the cache hierarchy and then main memory (cache hit or hit and cache miss or miss)
- Memory access terms read and write also used for caches
- If CPU attempts to read data or instructions and there is a cache read-miss, it will read from memory the cache line that contains the needed information and store it in the cache. This may stall the processor, while it waits for the slower memory
- CPU writing data to a cache (inconsistent values in main memory and cache). Two basic approaches to deal with it
  - In write-through caches, the line written in cache is also written in main memory
  - In write-back caches, the updated data in the cache is marked dirty and, when the cache line is replaced by a new cache line from memory, the dirty line is written to memory
- Another issue in cash design: where lines are stored? Cache mappings
  - fully associative cache: a new line can be placed at any location in the cache
  - direct mapped cache: each cache line has a unique location in the cache to which it will be assigned
  - n-way set associative cache: each cache line can be placed in n different locations in the cache

  Example: main memory of 16 lines with indexes $0 - 15$, cache of 4 lines with indexes $0 - 3$.
- When more than one line in memory can be mapped to several different locations in cache, one needs to decide which line should be replaced or evicted. The most common scheme is called last recently used. The cache has a record of the relative order in which the blocks have been used, the line with longer time is evicted.

Example of cash in programs: comparing two double loops in a matrix vector product.

1.4 Addressing von Neumann bottleneck: virtual memory

In some cases, all the instructions and data may not fit into main memory. This happens especially in multitasking systems, where many running programs must share the available main memory.

- Virtual memory was developed so that the main memory can function as a cache for secondary storage.
- It keeps in main memory only the active parts of the many running programs (locality).
- The parts that are idle are kept in a block of secondary storage called swap space.
- Virtual memory operates on blocks of data and instructions. These blocks are called pages.
- These pages are relatively large (page size ranges from 4 to 16 kilobytes) because secondary storage access can be hundreds of thousands slower than main memory access.
- If one assigns physical memory addresses to pages, when a program is compiled, then each page of the program will be assigned to one block of memory, and in the case of several programs running, some of them may want to use the same block of memory.
- To avoid this, when a program is compiled, its pages are assigned virtual page numbers. When the program is run, a table (page table) is created that maps the virtual page numbers to physical addresses (when managed by OS, it is guaranteed that memory used by one program does not overlap the memory used by another).
- A drawback is that it doubles the time needed to access a location in main memory (to execute an instruction in main memory, the executing program will have the virtual address of this instruction, and in order to translate it to physical address, it will need to find the page in memory that contains the instruction via the page table).
- If the required part of the page table is not in cache, one needs to load it from memory. After is loaded, one can translate the virtual address to a physical address and get the required instruction.
- Multiple programs can use the main memory at more or less he same time, and using a page table has the potential to significantly increase each programs overall run-time.
- To address this issue, processors have a special address translation cache called a translation-lookaside buffer, or TLB. It caches a small number of entries (typically $16 - 512$) from the page table in very fast memory.
- TLB hit (when virtual page number in TLB) and TLB miss (when virtual page number is not in TLB).
- If the page is not in memory (i.e., the page table does not have a valid physical address for the page and the page is only stored in disk, the attempt access is called page fault.
- The slowness of the disk accesses has additional consequences:
  - In virtual memory, disk access are so expensive that CPU caches handle write-misses with a write-back scheme. This can be handled by keeping a bit on each page in memory that indicates whether the page has been updated. It is has been updated, when evicted from main memory, it will be written to disk.
  - Since disk accesses are so slow, management of the page table and the handling of disk accesses can be done by OS. Thus, even though programmers do not directly control virtual memory, unlike CPU caches, which are handled by the system hardware, virtual memory is usually controlled by a combination of system hardware and OS software.

1.5 Addressing von Neumann bottleneck: low-level parallelism

1.5.1 Instruction-level parallelism

Instruction-level parallelism, or ILP, attempts to improve processor performance by having multiple processor components or functional units simultaneously executing instructions. There are two main approaches in ILP, both used in virtually all modern CPU's.

- pipelining, in which functional units are arranged in stages.

  To understand pipelining better, consider the following seven steps for the addition of two floating numbers

  1. Fetch operands
  2. Compare exponents
  3. Shift one operand
  4. Add
  5. Normalize results
  6. Round result
  7. Store result

  each taken a nanosecond.

  Now, assign each step of the addition of two floating numbers to a functional unit such

  - the output of one functional unit is the input of the next one.
  - Notice there is no advantage on a single statement but in a *for* loop, it is possible to execute simultaneously seven additions.
  - After the sixth step, the pipeline loop produces a result every nanosecond.

- multiple issue, in which multiple instructions can be simultaneously initiated.

  Multiple issue processors replicate functional units and try to simultaneously execute different instructions in a program. For example, consider two float point adders. While the first compute even additions, the second can compute odd additions.

  - If functional units are scheduled at compile time, the system is said to use static multiple issue.
  - If functional units are scheduled at run time, the system is said to use dynamic multiple issue (the processor that supports it is sometimes said to be superscalar).

  To make use of multiple issue, the system must find instructions that can be executed simultaneously. One important technique is speculation.

  - In speculation, the compiler or the processor makes a guess about an instruction, and then executes it on the basis of the guess, parallelizing some basic tasks of the instruction.
  - If the compiler does the speculation, it will usually insert code that test whether the speculation was correct, and if not takes corrective action.
  - If the hardware does the speculation, the processor usually stores the results of the speculative execution in a buffer.
    * When it is know that the speculation was correct, the contents of the buffer are transferred to registers or memory.
    * If the speculation was incorrect, the contents of the buffer is discarded and the instruction re-executed.

    While dynamic multiple issue systems can execute instructions out of order, in current generation systems, the instructions are still loaded in order and the results of the instructions are also written to registers and memory in the program-specified order.

    Optimizing compilers, on the other hand, can reorder instructions. This can have important consequences for shared-memory programming.

1.5.2 Hardware multithreading

A program with a long sequence of dependent statements offers few opportunities to ILP (e.g., Fibonacci).

- Thread-level parallelism, or TLP, attempts to provide parallelism through the simultaneous executions of different threads, so it provides a coarser-grained parallelism than ILP, that is the program units that are being simultaneously executed (threads), are larger or coarser than the finer-grained units (individual instructions).
- Hardware multithreading provides a means for systems to continue doing useful work when the task currently executed has stalled. Instead of looking for parallelism in the currently executed thread, it may make sense to simply run another thread. For this to be useful, the system must support very rapid switching between threads.
- In fine-grained multithreading, the processor switches between threads after each instruction, skipping threads that are stalled. While this approach has the potential to avoid wasted machine time due to stalls, it has the drawback that a thread that is ready to execute a long sequence of instructions may have to wait to execute every instruction.
- coarse-grained multithreading attempts to avoid this problem by only switching threads that are stalled waiting for a time-consuming operation to complete. This has the virtue that switching threads does not need to be nearly instantaneous. However, the processor can be idled on shorter stalls, and thread switching will also cause delays.
- Simultaneous multithreading, or SMT, is a variation of fine-grained multithreading. It attempts to exploit superscalar processor by allowing multiple threads to make use of the multiple functional units. If one designates preferred threads as the ones that have multiple instructions ready to execute, one can somewhat reduce the problem of thread slowdown.

2 Parallel hardware

Multiple issue and pipelining can clearly be considered to be parallel hardware, since functional units are replicated. However, this form of parallelism is not usually visible to the programmer. Nevertheless there is parallel hardware that is visible to the programmer in the sense that the programmer can modify the source code to exploit it. Consider this latter form of parallel hardware to be what is called parallel hardware.

Classification of computer architecture via Flynn's taxonomy: Characterizes a system according to the number of instruction streams and the number of data streams it can simultaneously manage.

2.1 SISD

Single instruction stream, single data stream, or SISD systems, executes a single instruction at a time and it can fetch or store one item of data at a time (e.g., classical von Neumann system).

2.2 SIMD

Single instruction, multiple data, or SIMD systems, is a parallel system that operates on multiple data streams by applying the same instruction to multiple data items.

- A SIMD system can be thought as having a single control unit and multiple ALUs.
- An instruction is broadcasted from the control unit to the ALUs, and each ALU either applies the instruction to the current data item, or it is idle.
- The requirement that all the ALUs execute the same instruction or are idle can seriously degrade the overall performance of a SIMD system.
- In a classical SIMD system, the ALUs must operate synchronously (they must wait until the next instructions is broadcasted before proceeding).
- ALUs have no instruction storage, so an ALU cannot delay execution of an instruction by storing for later execution.
- SIMD systems are ideal for parallelizing simple loops that operate on large arrays of data.
- Parallelism that is obtained by dividing data among the processors and having the processors all apply (more or less) the same instructions to their subsets of the data is called data-parallelism.
- SIMD parallelism can be very efficient on large data parallel problems, but often do not do very well on other types of parallel problems.
- Only widely produced SIMD systems are vector processors, or more recently, graphic processing units, or GPUs, and desktop CPUs are making use of aspects of SIMD computing.

2.2.1 Vector processors

Their key characteristic is that they can operate on arrays or *vectors* of data, while conventional CPUs operate on individual data elements or *scalars*.

Typically, they have the following characteristics:

* Vector registers: are capable of storing a vector of operands and operating simultaneously on their content. The vector length is fixed by the system, and can range from 4 to 128 64-bit elements.
* Vectorized and pipelined functional units: since vector operations act to each element in the operand vectors.
* Vector instructions: that operate on vectors rather than scalars, requiring only a single load, operation, and store for each block of prefixed vector-length elements, while conventional systems required a load, operation, and store for each element.
* Interleaved memory: the memory consists of multiple banks of memory, which can be accessed more or less independently. Since after accessing one bank, there will be a delay before it can be reaccessed again, different banks can be accessed much sooner. Vector elements distributed across multiple banks will display little or no delay in loading/storing successive elements.
* Typical vector systems provide special hardware to accelerate
  - Strided memory access: the program accesses elements of a vector loaded at fixed intervals.
  - Hardware scatter/gather: writing (scatter) or reading (gather) elements of a vector located at irregular intervals.

Advantages of vector processors for many applications

* They are very fast and very easy to use.
* Vectorizing compilers are quite good at identifying code that can be vectorized.
* Identify loops that cannot be vectorized, providing information to the user about why the loop cannot be vectorized. The user might be able to rewrite the loop to vectorize it.
* Vector systems have very high memory bandwidth, and every data item that is loaded is actually used, unlike cache-based systems that may not use every item in a cache line.

Drawbacks of vector processors

* They do not handle irregular data structures as well as other parallel architectures.
* There seems to be a very finite limit to their ability to handle ever larger problems (scalability).
* Current generations system scale by increasing the number of vector processors and not the vector length.
* Current commodity systems provide limited support for operations on very short vectors, while processors that operate on long vectors are very expensive.

2.2.2 Graphic processing units

* Real-time graphics application programming interfaces, or APIs, use points, lines and triangles to internally represent the surface of an object.
* They use graphics processing pipeline to convert an internal representation into an array of pixels that can be sent to a computer screen.
* Several stages of this pipeline are programmable.
* The behavior of the programmable stages is specified by functions called shader functions.
  - These are typically quite short.
  - They are implicitly parallel, since they can be applied to multiple elements in the graphics stream.
  - Application of shader functions o nearby elements often results in the same flow of control.
* GPUs can optimize performance by using SIMD parallelism. This is obtained by including a large number of ALUs (e.g., 80) on each GPU processing core.
* Processing a single image require very large amounts of data (hundreds of megabytes of data), and GPUs need to maintain very high rates of data movement, to avoid stalls on memory accesses. GPUs rely heavily on hardware multithreading.
* Even though the actual number of suspended thread states that can be stored for each executing thread depends on the number of the amount of resources (e.g., registers) needed by the shader function being executed, GPU systems can keep stored hundreds of threads.
* A drawback is that many threads processing a lot of data are needed to keep the ALUs busy, and GPUs may have relatively poor performance in small problems.
* GPUs are not pure SIMD systems. Although the ALUs on a given core do use SIMD parallelism, current generation GPUs can have dozens of cores, which are capable of executing independent instruction streams.
* GPUs are becoming popular for general high-performance computing, and several languages have been developed that allow users to exploit their power.

2.3 MIMD systems

Multiple instruction, multiple data, or MIMD systems, support multiple simultaneous instruction streams operating on multiple data streams.

* MIMD systems typically consists of a collection of fully independent processing units or cores, each of which has its own control unit and its own ALU.
* Unlike SIMD systems, MIMD system are asynchronous (processors can operate at their own pace).
* In many MIMD systems, there is no a global clock, and there may be no relation between the system times on two different processors.
* Unless the programmer imposes some synchronization, even if the processors are executing exactly the same sequence of instructions, at any given instant they may be executing different statements.

There are two principal types of MIMD systems: shared-memory and distributed-memory.

* In a shared-memory system
  - a collection of autonomous processors is connected to a memory system via an interconnection network, and each processor can access each memory location.
  - processors usually communicate implicitly by accessing shared data structures.
  - The most widely available shared-memory systems use one or more multicore processors (each has multiple CPUs or cores on a single chip; typically the cores have private level L1 caches, while other caches may or may not be shared between the cores).
  - In shared-memory systems with multiple multicore processors, the interconnect can either connect all the processors directly to main memory (uniform memory access, or UMA system), or each processor can have a direct connection to a block of main memory, and the processors can access each other's blocks of main memory through special hardware built into the processors (nonuniform memory access, or NUMA system).
  - UMA systems are usually easier to program, since the programmer does not need to be aware about different access times for different locations.
  - This advantage can be offset by the faster access to the directly connected memory in NUMA systems (which have the potential to use larger amounts of memory than UMA systems).
* In a distributed-memory system,
  - each processor is paired with its own *private* memory
  - the processor-memory pairs communicate over the interconnection network.
  - Processors usually communicate explicitly by sending messages or by using special functions that provide access to the memory of another processor.
  - The most widely available distributed-memory systems are called clusters.
  - Clusters are composed of a collection of commodity systems (e.g., PCs), connected by a commodity interconnection network (e.g., ethernet). The nodes (the individual computational units joined together by the communication network) of these systems, are usually shared-memory systems with one or more multicore processors.
  - To distinguish such systems from pure-distributed systems, they are called hybrid systems.
  - Nowadays, it is understood that a cluster will have shared-memory nodes.
  - The grid provides infrastructure necessary to turn large networks of geographically distributed computers into a unified distributed-memory system. In general, such a system will be *heterogeneous* (the individual nodes may be built from different types of hardware).

2.4 Cache coherence

- CPU caches are managed by hardware; programmers do not have direct control over them.
- Consider a shared-memory system (each core with its own private data cache)
  - As long as the cores only read shared data, there is no problem.
  - However, assume variable x is in the shared memory and it is loaded in cores 0 and 1 caches. Suppose variable x is utilized by core 1 for computing variable z (that is also in its private cache). If core 0 updates variable x in its own cache before core 1 does, unless variable x is evicted from core 0 cache and loaded into core 1 cache, the content of variable z in cache 1 is unpredictable.
  - This unpredictable behavior will occur regardless the CPU writing data policy.
    - If the system is using a write-through policy, the main memory will be updated with core 0 cache value of variable x, but that will have no effect on the value of variable x in core 1 cache.
    - If the system uses a write-back policy, the new value of x in core 0 cache will not even be available to core 1 when it updates variable z.
  - This is called the cache coherence problem. The caches for single processor systems provide no mechanism for insuring that when the caches of multiple processors store the same variable, an update by one processor to the cache variable is *seen* by the other processors.
  - There are two approaches to insure cache coherence
    - snooping cache coherence: when the cores share an interconnect, any signal transmitted on it can be *seen* by the cores connected to the interconnect.
    - If core 0 updates x in its cache, the fact that x has been updated (but not the new value) in core 0 cache is broadcast throughout the interconnect, and if core 1 is *snooping* the interconnect, it can flag its copy of variable x in its cache as invalid.
    - Snooping works for both write-through and write-back caches. With write-through cache there is no need for additional traffic on the interconnect, since each core can simply *watch* for writes. With write-back caches, an extra communication is needed, since updates to the cache do not get immediately send to main memory.
    - Unfortunately, in large networks broadcast every time a variable is updated is expensive, and it will cause performance degrade.
    - directory-based cache coherence: in these protocols, a data structure called a directory stores the status of each cache line.
    - Typically, this data structure is distributed (each core-memory pair might be responsible for storing part of the structure that specifies the status of the cache in its local memory).
    - When a line is read into core 0 cache, the directory entry corresponding to that line would be updated indicating that core 0 has a copy of the line.
    - When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variables's cache line in their caches are invalidated.
    - There will be substantial additional storage required for the directory, but when a cache variable is updated, only cores storing that variable need to be contacted.

2.4.1 False sharing

CPU caches are implemented in hardware, so they operate on cache lines, not individual variables.

This can have disastrous consequences for performance.

* Suppose a function $f(x, y)$ is repeatedly called in an iterative procedure (on x externally m times, and on y internally n times) and its results are added into a vector $v(x)$.
* One can parallelize this task by dividing the external loop among c cores of a shared-memory system, and by assigning m/c iterations to each core.
* Assume the parts of vector v corresponding to cores 0 and 1 are in the same cache line.
* When cores 0 and 1 simultaneously execute their codes, each time they perform the addition instruction, the line will be invalidated, and the next time core 0 or core 1 performs an addition, it will have to fetch the updated line from memory.
* A large percentage of the addition assignments will access main memory, in spite the fact core 0 and core 1 never access each other's elements of vector v.
* This is called false sharing, because the system is behaving as if the elements of v were being shared by the cores.

2.4.2 Shared-memory versus distributed-memory

Why all MIMD systems are not shared-memory, since most programmers find the concept of implicitly coordinating the work of processors through shared structures more appealing than explicitly sending messages?

* There are several other issues.
* The principal hardware issue is the cost of scaling the interconnect.
* As the number of processors connected to a interconnect bus increased, the chance there will be conflicts over access to the bus increase dramatically.
* Distributed-memory interconnects are relatively inexpensive.
* Distributed-memory systems are often better suited for problems requiring vast amounts of data or computation.

3 Parallel software

It is no longer possible to rely on hardware and compilers o provide a steady increase in application performance. It is needed to learn how to write applications that exploit shared- and distributed-memory architecture.

Some terminology:

- When running shared-memory programs, one starts a single process and fork multiple threads, so one talks about threads carrying out tasks.
- When running distributed-memory programs, one start multiple processes, so one talks about multiple processes carrying out tasks.
- When the discussion applies equally well to shared-memory and distributed-memory systems, one talks about processes/threads carrying out tasks.

3.1 Pacheco's book caveats

- Only discuss software for MIMD systems (in particular, non-GPUs systems).
- The topics covered give only some idea of the issues when writing parallel software.
- It mainly focus on what is called single program, multiple data, or SPMD.
- Instead of running a different program on each core, SPMD programs consist of a single executable that can behave as if it were multiple different programs through the use of conditional branches.
- For example, SPMD can readily implement parallelism (a program is task parallel if it obtains its parallelism by dividing tasks among threads or processes).

3.2 Coordinating the processes/threads

In very few cases, obtaining parallel performance is trivial.

As an example, consider the case that one only needs to assign elements of the data to different processes. The programmer only needs to

1. Divide the work among the processes/threads
   (a) in such a way that each process/thread get roughly the same amount of work (load balancing)
      - This is a concern in situations when the amount of work is not known in advance by the programmer (e.g., a tree search)

      Programs that can be parallelized by simply dividing the work among the processes/threads are called embarrassingly parallel.

   (b) in such a way that the amount of communication is minimized

   When a program is not embarrassingly parallel, one needs to coordinate the work of the processes/threads. In these programs, one usually needs to
2. Arrange for the processes/threads to synchronize
3. Arrange for communication among the processes/threads

3.3 Shared-memory

In shared-memory programs, variables can be shared or private.

- Shared variables can be read or written by any thread.
- Private variables can only be accessed by one thread.
- Communication among the threads is usually done by shared variables (implicit rather than explicit).

3.3.1 Dynamic and static threads

In different environments shared-memory programs use the following paradigms

- dynamic threads: there is often a master thread and at any given instant a (possibly empty) collection of worker threads.
  - The master thread typically waits for work requests.
  - When the request arrives, the master forks a worker thread, the thread carries out the request, and when the thread completes the work, it terminates and joins the master thread.

  This paradigm makes efficient use of system resources (resources required by any thread are only being used while the thread is actually running).
- static threads:
  - all of the threads are forked after any needed setup by the master thread and the threads run until all the work is completed.
  - After the threads join the master thread, the master thread may do some cleanup (e.g., free memory) and the it also terminates.

  In terms of resource usage, this may be less efficient (if a thread is idle, its resources cannot be freed).

However, forking and joining threads can be fairly time-consuming operations. So if the necessary resources are available, the static thread paradigm has the potential for better performance than the dynamic paradigm.

It also has the virtue that it is closer to the most widely used paradigm for distributed-memory programming, so part of the mindset that is used for one type of system is preserved for the other.

One often uses the static thread paradigm.

3.3.2 Nondeterminism

In any MIMD system in which the processors execute asynchronously it is likely there will be nondeterminism.

A computation is nondeterministic if a given input can result in different outputs.

- If multiple threads are executing independently, the relative rate at which they will complete statements varies from run to run, and hence the results of the program may be different from run to run.
- In addition, the output of one thread could be broken by the output of another thread.
- Because the threads are executing independently and interacting with the OS, the time it takes for one thread to complete a block of statements varies from execution to execution, so the order in which these statements complete cannot be predicted.
- In many cases nondeterminism is not a problem, however there are many cases in which nondeterminism, especially for shared-memory programs, can be disastrous, because it can easily result in program errors.
- Consider the case when different threads attempt to simultaneously update a variable.

When threads or processes attempt to simultaneously access a resource, and these can result in an error

- it is said that the program has a race condition, because the threads or processes are in a *horse race* (the outcome of the computation depends on which thread wins the race).
- A block of code that can only be executed by one thread at a time is called a critical section.
- Programmers' job is to insure mutually exclusive access to the critical section (that if one thread is executing the code in the critical section, then the other threads are excluded).

Mechanisms for insuring mutual exclusion

- Mutex
  - The most commonly used mechanism for insuring mutual exclusion is a mutual exclusion lock or mutex or lock.
  - A mutex is a special type of object that has support in the underlying hardware.
  - The basic idea is that each critical section is protected by a lock.
  - Before a thread can execute the code in the critical section, it must *obtain* the mutex by calling a mutex function, and, when it is done executing the code in the critical section, it should *relinquish* the mutex by calling an unlock function.
  - While one threads *owns* the lock (that is, has returned from a call to the lock function), but has not yet called the unlock function, any other thread attempting to execute the code in the critical section will wait in its call to the lock function.
  - Since a mutex enforces serialization of the critical section, one should reduce the number of critical sections as much as possible.
- Busy-waiting
  - In busy-waiting, a thread enters a loop whose sole purpose is to test a condition (the thread can be very busy waiting for the condition).
  - It is simple to understand.
  - It is very wasteful of system resources, because even when a thread is doing no useful work, the core running the thread will be repeatedly checking to see if the critical section can be entered.
- Semaphores
  - Similar to mutexes, although the details for their behavior are slightly different.
  - There are some types of thread synchronization that are easier to implement with semaphores than mutexes.
- Monitors
  - Provide mutual exclusion at a somewhat higher-level.
  - It is an object whose methods can only be executed by one thread at a time.
- Transactional memory
  - Critical sections in shared-memory programs should be treated as transactions.
  - Either a thread successfully completes the critical section or any partial results are rolled back and the critical section is repeated.

3.3.3 Thread safety

Parallel programs can call functions developed for use in serial programs, and most of the time there will not be any problems, but there are notable exceptions.

- Functions that utilize *static* variables.
- C local variables are declared inside a function and are allocated from the system stack.
- Since each thread has its own stack, ordinary C variables are private.
- However, static variables declared in a function persist from one call to the next one.
- Effectively, static variables are shared among the threads that call the function, and this can have unexpected and unwanted consequences.

This kind of functions if they are used in a multithreaded program can produce errors or unexpected results. Such a function is not thread safe.

When a block of code is not thread safe, it is usually because different threads are accessing shared data.

Even though many serial functions can be used safely in multithreaded programs, programmers need to be wary of functions that were written exclusively for use in serial programs.

3.4 Distributed-memory

In distributed-memory programs, the cores can directly access only their own, private memories.

- There are several APIs that are used.
  - One of the first things to note regarding distributed-memory APIs is that they can be used with memory-shared hardware.
  - It is perfectly feasible for programmers to logically partition shared-memory into private address spaces for the various threads, and a library or compiler can implement the communication that is needed.
  - Distributed-memory programs are usually executed by starting multiple processes rather than multiple threads. This is because, typical threads of execution may run on independent CPUs with independent OS, and there might be no software infrastructure for starting a single distributed process and having the process fork one or more threads on each node of the system.
- However, by far the most widely used is message-passing.
  - A message-passing API provides (at a minimum) a *Send* and a *Receive* functions.
  - Processes typically identify each other by ranks in the range $0, 1, \cdots, p-1$, where p is the number of processes.
  - The processes branch depending on their ranks.
  - The processes use the same executable, but carrying out different actions (usually depending on their ranks).
  - The variables that are communicated via messages, even though they have the same name, they refer to different blocks of memory on the different processes.
  - In most implementations of message-passing API's, allow all processes access to *stdout* and *stderr*, even if the API does not explicitly provide for this.
  - There are several possibilities for the exact behavior of he *Send* and the *Receive* functions.
  - The simplest behavior is for the call to *Send* a block until the call for *Receive* starts receiving the data.

    Send

    * This means that the process calling *Send* will not return from the call until the matching call to *Receive* has started.
    * Alternatively, the *Send* function may copy the contents of the message into storage that it owns, and then it will return as soon as the data is copied.

    Receive

    * The most common behavior for the *Receive* function is for the receive process to block until the message is received.

    There are other possibilities for *Send* and *Receive*.
  - Typical message-passing APIs provide a variety of additional functions.
    * broadcast: a single process transmit the same data to all the processes.
    * reduction: results computed by the individual processes are combined into a single result.
    * There may also be special functions for managing processes and communicating complicated data structures.
  - Even though message-passing is a very powerful and versatile API for developing parallel programs and that virtually all powerful computers in the world use it, there is a huge amount of detail that programmers need to manage (very low level). There have been many attempts to develop other APIs.

- There are others less frequently used.
  - One-sided communication, or remote memory access: In message-passing any communication requires the explicit participation of two process (e.g., one process must call a send function and the send must be matched by another process' call to receive function). In one-sided communication, a single process calls a function, which updates either
    * a local memory with a value from another process, or
    * remote memory with a value from the calling process

    This can simplify communication , and significantly reduces the cost of communication by eliminating the overhead associated with synchronizing two processes, as well as, eliminating the overhead of one of the function calls.

    These advantages are hard to realize in practice
    * if one process is copying a value into the memory of other process, the first process must have some way of knowing when it is safe to copy, since it will overwrite some memory location (this can be solved by synchronizing both processes before the copy).
    * In addition, the second process must also have some way of knowing when the memory location has been updated (this can be solved by another synchronization or flag variable that the first process sets after it completed the copy). If this is the case, the second process may need to poll the flag variable to determine that the new value is available.

    These problems can considerably increase the overhead associated with transmitting a value.

    In addition, since there is no explicit interaction between the two processes, remote memory operations can introduce errors that are very hard to track down.
  - Partitioned global address space languages, or PGAS languages: provide some of the mechanisms of shared-memory programs. Private variables are allocated in the local memory of the core on which the process is executing, and the distribution of the data in shared data structures is controlled by the programmer.
- Programming hybrid systems: It is possible to program systems such as clusters of multicore processors using a combination of a shared-memory API on nodes and a distributed-memory API for internode communication. However, this is usually only done for programs that require the highest possible levels of performance, since this *hybrid* API makes program development extremely difficult.

4 Input and output

- Parallel I/O, in which multiple cores access multiple disk or other devices, is a large subject (not covered).
- Vast majority of programs developed do very little in terms of I/O.
- The amount of data they read and write is quite small and easily managed by standard C I/O functions (serial functions (*printf, fprinft, scanf, fscanf*).
- Threads that are forked by a single process do share *stdin*, *stout*, *stderr*.
- However, when multiple threads attempt to access one of theses functions, the outcome is nondeterministic.
  - When calling to *printf*, one would like the output appear on the console of a single system, the system on which the program started (and it is what the vast majority of systems do), but there is no guarantee of that. It may happen that only one process (or no processes) has/have access to *stdout* or *stderr*.
  - What should happen with calls to *scanf* when running multiple processes is less obvious. Should be the input be divided among the processors or should only a single process to call *scanf*? The vast majority of systems allow at least one process to call *scanf* (usually the master), while some systems allow more processes or none at all.
  - When multiple processes can access *stdin, stdout, stderr*, the distribution of the input and output are nondeterministic.
    - For output, the data will appear in a different order each time the program is run, or even worse, the output of one process may be broken up by the output of another process.
    - For input, the data read by each process may be different on each run, even if the same input is used.

  To partially address these issues, we will make the following assumptions:
  - In distributed-memory programs, only process 0 will access *stdin*.
  - In shared-memory programs, only the master thread or thread 0 will access *stdin*.
  - In distributed-memory and shared-memory programs, all the processes/threads can access *stdout* and *stderr*.
  - However, because of the nondeterministic order of output to *stdout*, in most cases a single process/thread will be used for ll output to *stdout* (an exception will be output for debugging, and in this situation often multiple processes/threads can write to *stdout*).
  - Only a single process/thread will attempt to access any single file other than *printf, fprinft, scanf, fscanf* (each process/thread can open its own, private file for reading and writing, but no two processes/threads will open the same file).
  - Debug output should always include the rank or id of the process/thread that is generating the output.

5 Performance

The main purpose in writing parallel programs is usually increased performance.

5.1 Speedup and efficiency

- Suppose T_{serial} is the serial run-time of a program.
- Suppose $T_{parallel}$ is the parallel run-time of a program.

The best one can hope is to equally divide the work among the cores while at the same time introducing no additional work for the cores.

- If one succeed in doing this, and the program is run with p cores, one thread or process on each core, then the program will run p times faster than the serial program and

$$T_{parallel} = T_{parallel}(p) = \frac{T_{serial}}{p} \quad \text{(serial speedup)}.$$

- In practice, it is unlikely to get linear speedup because the use of multiple processes/threads almost invariably introduces some overhead.
  - Shared-memory programs will almost always have critical sections, which will require the use of some mutual exclusion mechanism (e.g., mutex). The call to the mutex functions are overhead that is not present in the serial program, and the use of the mutex forces the parallel program to serialize execution in the critical section.
  - Distributed-memory programs will almost always need to transmit data across the network, which is usually much slower than the local memory access. Serial programs will not have these overheads.
  - Furthermore, it is likely that the overhead will increase with the umber of processes or threads (more threads mean more threads need to access a critical section and more processes mean more data need to be transmitted across the network).
- Two statistics:
  - The speedup of a parallel program

$$S = S(p) = \frac{T_{serial}}{T_{parallel}}.$$

    The ideal linear speedup is $S = p$. Furthermore, as p increases, one expect S to become a smaller and smaller fraction of the ideal, linear speedup p. Another way of saying it is that S/p will probably get smaller and smaller as p increases.
  - The efficiency of the parallel program

$$E = E(p) = \frac{S}{p} = \frac{T_{serial}}{pT_{parallel}}.$$

  Actually, if n is the problem size then

$$T_{serial} = T_{serial}(n), \quad T_{parallel} = T_{parallel}(p, n), \quad S = S(p, n), \quad E = E(p, n).$$

- Many parallel programs are developed by dividing the work of serial programs among the processes/threads and adding in the necessary parallel overhead. If $T_{overhead}$ is this parallel overhead, it is often the case that

$$T_{parallel} = \frac{T_{serial}}{p} + T_{overhead}.$$

  Furthermore, as n increases, $T_{overhead}$ often grows more slowly than T_{serial}. When this is the case, $S(p, n)$ and $E(p, n)$ will increase.

What values of T_{serial} should be used when reporting speedups and efficiencies?

- Some say T_{serial} should be the run-time of the fastest program on the fastest processor available.
- In practice, one uses a serial program on which the parallel program was based and run it on a single processor of the parallel system.

5.2 Amdahl's law

It roughly says:

Unless virtually all of a serial program is parallelized,
the possible speedup is going to be very limited,
regardless of the number of cores available.

- If a fraction r of a serial program remains unparallelized, then Amdahl's law says that it is not possible to get an speedup better than $\frac{1}{r}$.
  - Suppose one is able to parallelize $(1-r)\%$ of the serial program
  - Suppose the parallelization is perfect, that means, regardless of the number of cores used, the speedup of this part of the program is p.
  - If the serial run-time is T_{serial}, then
    - the run-time of the parallelized part will be $\frac{(1-r)}{p}\,T_{serial}$, and
    - the run-time of the unparallelized part will be $r\,T_{serial}$.
  - The overall parallel run-time will be

$$T_{parallel} = \frac{(1-r)}{p}\,T_{serial} + r\,T_{serial} = \left(\frac{(1-r)}{p} + r\right)\,T_{serial},$$

    and the speedup will be

$$S = \frac{T_{serial}}{\left(\frac{(1-r)}{p} + r\right)\,T_{serial}} = \left(\frac{1}{\frac{(1-r)}{p} + r}\right) \leq \frac{1}{r}.$$

- Should we give up? Well, no.
  - Amdahl's law does not take into consideration the problem size.
  - Usually, as the problem size increases, the inherent serial fraction of the program decreases in size.

- (Gustafson's law) Suppose the execution time of a program running on a parallel system can be split into two parts:

$$T_{parallel} = T_{\text{serial part}} + T_{\text{parallel part}},$$

where $T_{\text{serial part}}$ is the part that does not benefit from the increasing number of processors, and $T_{\text{parallel part}}$ is the part that benefit from it.

Define the relative times R of the serial and parallel parts as

$$R_{\text{serial part}} = \frac{T_{\text{serial part}}}{T_{parallel}}, \quad R_{\text{parallel part}} = \frac{T_{\text{parallel part}}}{T_{parallel}}.$$

Notice that

$$R_{\text{serial part}} + R_{\text{parallel part}} = 1.$$

Hypothetically, when running a program on a serial system (only one processor), the serial part takes $T_{\text{serial part}}$, while the parallel part now takes $p\,T_{\text{parallel part}}$.

The execution time on the serial system is:

$$T_{serial} = T_{\text{serial part}} + p\,T_{\text{parallel part}}.$$

Using T_{serial} as the baseline, the speedup for the parallel system is:

$$S = \frac{T_{serial}}{T_{parallel}} = \frac{T_{\text{serial part}} + p\,T_{\text{parallel part}}}{T_{parallel}} = \frac{T_{\text{serial part}}}{T_{parallel}} + p\,\frac{T_{\text{parallel part}}}{T_{parallel}} = R_{\text{serial part}} + p\,R_{\text{parallel part}},$$

and hence,

$$S = p + (1-p)\,R_{\text{serial part}},$$

or equivalently,

$$S = 1 + (p-1)\,R_{\text{parallel part}}.$$

5.3 Scalability

- Suppose one runs a parallel program for a problem of size n, with a fixed number p of processes/threads and a fixed input size, and one obtains and efficiency E.
- Suppose now that the number of processes/threads used by the program is increased to $p^{'} = k\,p$, $k > 1$.
- If one finds a corresponding rate of increase x in the problem size $n^{'} = x\,n$, so that the program always has efficiency $E^{'} = E$, then the program is scalable.
- In other words, if one increases the problem size at the same rate that one increases the number of processes/threads, and the efficiency is unchanged, then the program is scalable.
  - If when one increases the number of processes/threads, one can keep the efficiency *without* increasing the problem size, the program is said to be strongly scalable.
  - If one can keep the efficiency fixed by increasing the problem size at the same rate as one increases the number of processes/threads, then the program is said to be weakly scalable.

5.4 Taking timings

How to find T_{serial} and $T_{parallel}$?

- There are a lot of different approaches, and with parallel programs the details may depend on the API.
- However, there are a few general observations one can make that may make things a little easier.
  - Why taking timings? Is one taking timings differently in different situations?
    - To determine is the program is behaving as intended (in this case one usually needs very detailed information of much time the program is spending in each section of it).
    - Once completed the development of the program, one is often interested in how good its performance is (quantity reported by a single value).
  - Taking timings for what part of the program?
    - Usually, one is not interested in the time that elapses between the program's start and the program's finish.
    - Usually, one is interested only in some part of the program.
    - The Unix *time* command reports the time taken to run a program from start to finish.
  - One is not interested in CPU time.
    - Reported by the C function *clock* (the total time the program spends in code executed as part of the program).
    - It would include: the time for code one has written, the time the OS spends in library functions calls (e.g., *pow, sin*), the time the OS spends in functions called by the program (e.g., *printf, scanf*), and it will not include the time the program was idle, and the latter could be a problem.
    - For example, in a distributed-memory program, a process that calls a receive function may have to wait for the sending process to sleep while it waits. This idle time would not be counted as CPU time, since no function that has been called by the process is active. However, it should count in the evaluation of the overall run-time, since it may be a real cost of the program.
    - In an article reporting the run-time of a parallel program, the reported time is usually *wall clock* (the time that has elapsed between the start and finish of the execution of the code the user is interested in).
- There might be an issue with the resolution of the timer function.
  - It is the unit of measurement on the timer.
  - It is the duration of the shortest event that can have a nonzero time.
  - Some timer functions have resolutions in milliseconds and others in different units.
- One is usually interested in a single time
  - the time that has elapsed from when the first process/thread began the execution of the code to the time the last process/thread finished execution of the code.
  - Often one cannot obtain this exactly since there may not be correspondence between the clock on one node and the clock on another node.

- Using barriers.
  - The execution of a barrier function approximately synchronizes all of the process/threads.
  - One would like for all the processes/threads to return from the call simultaneously, but such a function usually can only guarantee that all the processes/threads have started the call when the first process/thread returns.
  - One then executes the code as before and each process/thread finds the time it took.
  - Then all the processes/threads call a global maximum function, which returns the largest of the elapsed times, and process/thread 0 prints it out.
- One also needs to be aware of the variability of timings.
  - When one runs a program several times, it is extremely likely that the elapsed time will be different for each run.
  - This will be true even if each time one runs the program one uses the same input and the same systems.
  - Since it is unlikely that some outside event could actually make a program run faster than its best possible run-time, one usually reports the minimum time.
- Running more than one thread per core can cause dramatic increases in the variability of timings.
  - More importantly, if one runs more than one thread per core, the system will have to take extra time to schedule and deschedule cores, and this will add to the overall run-time.
  - Therefore, one rarely runs more than one thread per core.
- Since most of the programs will not be designed for high-performance I/O, one usually does not include I/O in the reported run-times.

6 Parallel program design

How to parallelize a serial program?

- In general one needs to divide the work among the processes/threads so that each process gets roughly the same amount of work and communication is minimized.
- In most cases, one also needs to arrange for the processes/threads to synchronize and communicate.
- Unfortunately there is no mechanical process that one can follow.

Foster's methodology

1. *Partitioning*: Divide the computation to be performed and the data operated on by the computation into small tasks. The focus should be on identifying tasks that can be executed in parallel.
2. *Communication*: Determine communication needs to be carried out among the tasks identified in step 1.
3. *Agglomeration or aggregation*: Combine tasks and communications identified in step 1 into larger tasks (e.g., if task A has to be executed before task B, it may make sense to combine them into a single composite task).
4. *Mapping*: Assign the composite tasks identified in step 3 to processes/threads. This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.

7 Writing and running parallel programs

Shared-memory systems:

- In smaller shared-memory systems, there is a single running copy of the OS, which ordinarily schedules the threads on the available cores.
- On these systems, shared-memory programs can usually be started using either an IDE or the command line.
- Once started, the program will typically use the console and the keyword for input from *stdin* and output to *stdout* and *stderr*.
- On larger systems, there may be a batch scheduler; that is a user requests certain number of cores, and specifies the path to the executable and where input and output should go (typically to files in secondary storage).

Distributed-memory systems (and hybrid systems):

- Typically there is a host computer that is responsible for allocating the nodes among the users.
- Some systems are purely batch systems, which are similar to shared-memory batch systems.
- Others allow users to check out nodes and run jobs interactively.
- Since job startup often involves communicating with remote systems, the actual startup is usually done with a script (e.g., MPI programs are usually started with a script called *mpirun* or *mpiexec*).

8 Assumptions

As noted earlier

- We will focus on homogeneous MIMD systems (systems in which all nodes have the same architecture).
- The programs will be SPMD (we will write a single program that can use branching to have multiple different behaviors; and we will assume that the cores are identical but that they operate asynchronously).
- We always run at most one process or thread of our program on a single core, and we will often use static processes or threads (in other words, we will start the processes or threads more or less at the same time, and when they are done executing, we will terminate them at more or less the same time).
- We will use parallel extensions to the C language.
- We will use *gcc* compiler or some extension of it (e.g., *mpicc*).
- We will start programs from the command line.
- We will usually use the following options for the compiler:
  - $-g$ (creates information that allows to use a debugger).
  - $-Wall$ (issue lots of warnings).
  - $-o\ \langle outfile \rangle$ (put the executable in the file name *outfile*).
  - $-O2$ (for telling the compiler to optimize the code when timing programs).