**Why not push the clock faster?**

Speed/power tradeoff – the amount of power needed to increase a bit of speed is too big

Underclocking a core by 20% saves 50% of power while sacrificing only 13% performance

**How to program multicore processors?**

Will compilers do the job?

* No
* Even for sequential programming we need to write code carefully if we want to get performance and scalable program (data size and locality)

Main challenge

* Write scalable program that:
  + Keep the efficiency level as Data increases
  + Keep the efficiency level as more cores are available

**Main goal of parallel computing**

* Scalable (resource-aware) computing
* Resources in computing:
  + sets of (processor + memory + interconnection)
  + understand the trend past-present-future
  + be prepared for heterogeneity: general-purpose and attached devices
* Performance evaluation
  + performance and efficiency measures
  + scalability analisys

A diagram of a problem

Description automatically generated with medium confidence

**Scientific Computing**

Paralel computing alreading existed before the multicore era.

But, back then it was used in a quite specific contezt – Scientific Computing.

A diagram of a problem

Description automatically generated with medium confidenceNow, any computer programmer must be aware of it.

Scientific method: classic approach (first image)

Modern scientific method (second image)

**Parallel Computing**

Why should we use parallel computing?

* Possibility of solving bigger problems and with more realistic representation (higher accuracy/detail)
  + example: weather forecast for more days and with more accuracy
* Reduce development costs
* have higher freedom to “explore” alternatives

**Performance**

Performance metrics

* MIPS
  + million instrucitons per second
  + for integer operations
    - also called “meaningless indicator of performance” MIP
* FLOPS
  + floating-point operations per seconds
  + for scientific applications

Peak performance (Rpeak Top500)

* Related to CPU speed

Maximum performance (Rmax Top500)

* Maximum performance for a given algorithm

Nmax – problem size to achieve Rmax

Sustained performance

* Computer performance depends on several factors: I/O speed, data access pattern, memory hierarchy
* The relevant performance is the one that resuts from the real execution of an algorithm
* The sustained performance depends also on the algorithm design
  + An implementation compatible with the computer arquitcture can achieve the same performance (sustained) for a wider range of input data
* Example: matrix multiplication algorithm

**Programming multicore processors**

* Even for sequential programming we need to do explicit memory management to get performance and scalable programs

**Parallelism and Amdahl law**

* In an application there is always a part that cannot be aprallelized
* Amdahl Law
  + Let “s” be the piece of work that is sequential. (1-s) will be the piece of work that can be parallellized
  + P – number of processors
* A black text on a white background

  Description automatically generated with low confidenceEven if the aprallel part is perfectly scalable, the performance (Speedup) is limited by the sequential part
* Amdahl law (example 1st ppt, slide 32)
  + A picture containing font, line, diagram, white

    Description automatically generatedThis law imposes a limit for the speedup that can be optained with P processors
  + A picture containing font, text, line, handwriting

    Description automatically generatedex. if the total execution time of an algorithm is 93s and the sequential time susceptible of parallelization is 90s, then:
    - (1-s) = 90/93 = 0.968 -> 96.8% of code can be parallelized
    - s = 1-0.968 = 0.032 -> 3.2% of code is inheretly sequential
  + Code susceptible of parallelization
    - is the part of the code that executes with Speedup=P if it runs on P processors
  + Code inheretly sequential
    - is the part of the code that cannot be parallelized, such as data input/output, variable initialization, etc.
      * If P -> Infite, the Speedup -> 1/s
    - For the last example, the maximum speedup woll be:
      * Speedup(max) = 1/0.032 = 31.25
  + In conclusion:
    - whatever the most number of proceesors used the processing time will not be less than 1/31.25
* Important consideration taken from Amdahl Law:
  + it allows to have a realistic expectation, for a given algorithm, about what we can obtain with the parallel approach
  + it shows that to achieve higher Speedups it is necessary to reduce or eliminate the algorithm sequential blocks
  + it also gives a comparison metric to measure parallelizability of several algorithm for the same problem

A picture containing text, diagram, font, line

Description automatically generated**Functional Parallelism**

Independ tasks execute different operations on different data sets

* Instructions 1 and 2 are independent
* Instructions 3 and 4 are dependent from 1 and 2 but are independent from each other

Data dependency graph

* A diagram of a data analysis

  Description automatically generated with low confidencedirectional acyclic graph (bottom image)
  + edges: functional dependencies
  + vertices: tasks

A picture containing font, text, screenshot, graphics

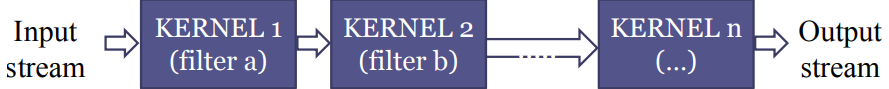
Description automatically generated**Data parallelism**

Independent tasks execute same operation over different data

Example (image):

* Vector elements can be added in an independent way. The sum operation can be applied simultaneously over the different vector elements b and c.

**Streaming**

* ****To process streams of data
  + Divide the process in steps
  + the number of steps limits the speedup
* To process multiple streams of data
* examples:
  + real time data analysis
  + real time decision making support

**Ways of extraction parallelism**

Last three topics we say:

* functional parallelism
* data parallelism
* streaming

**Parallel Programming models**

* Shared memory model
* Distributed memory model
* MapReduce model

**Shared Memory model**

* Each proceesor (or core) executes a thread
* Threads interact by shared variables
* Fork/join parallelism
  + number of fork/joins influences performance
  + initially only master thread is active
    - master thread executes sequential code
  + fork
    - master thread creates or awakens additional threads to execute parallel code
  + join
    - at end of parallel code created threads die or are suspended
* Threads
  + each thread has its own process state, but share global variables defined by the master thread
* Parallel “for” loops
  + C programs often express data-parallel operations as “for” loops
  + a multithreaded program can split the “for” loop to execute concurrently

**Race condition**

* when one process may “race ahead” of another and not see its change to a shared variable
* a date race occurs when two or more threads can modify the same memory location at the same tim

**Critical Section**

* a portion of code that only a thread can can execute at a time
* we dont a critical section by putting the pragma:
  + “#pragma omp critical” before a block of code

**False sharing**

* when 2 or more threads access different data on the same cache line (read/write)
* example: access close poitions of a global vector

**Distributed Memory Model**

Parallel program = a set of tasks executing concurrently

* Task
  + sequential program (von Neumann model)
  + local memory
  + a set of I/O ports
* A picture containing text, diagram, screenshot, circle

  Description automatically generatedTasks interact by send messages through the communication channels

Methodology to develop parallel programs:

* Problem partitioning
* communication patterns
* agglomeration
* mapping

**MapReduce model**

* MapReduce is a programming model for writing applications that cna process Big Data in parallel on multiple nodes
* MapReduce provides analytical capabilities for analyzing huge volumes of complex data
* Why MapReduce?
  + commom approach of centralized processing results in huge processing time when considering big data
  + distributed/parallel processing is the solution but it is harder to manage
  + However, for a specific processing problem of:
    - applying a given function to all data elements
    - combining those results using also user defined function
    - a programming model can be defined: MapReduce
  + MapReduce algorithm was developed by google to tackle the Page Rank problem.
* Composed by two essential steps:
  + map: (data parallel operation)
    - applies a user defined function over a set of key-value pairs, generating other set of key-value pairs
  + reduce: (functional parallel operation)
    - applies a user defined reducer function to a set of key-value pairs, where data can be aggregated, filtered and combined in a number of ways
* in map-reduce, job is usually split from input data set into independent chunks, which are processed by the map tasks in a completely parallel (nidependent) manner. The framework then sorts the outputs of the map operation, which are then supplied to the reduce tasks

**Classification of the operations**

* Sequential operations (ex. first ppt, page 77)
  + operations that require some effort to be parallelized
* parallel operations (ex. first ppt, page 78)
  + operations that are embarrassingly parallel

**Memory hierarchy**

* The CACHE memory serves of interface between the processor and the main memory
* Explores the temporal and spatial location of data

**Memory technology**

* Faster memories -> more expensive per bit because require more area per bit

**Cache memory**

* Facts:
  + execution time = clock cycles executing user code + clock cycles for data transfer between cache and main memory
  + for many years the increase rate of processor speed was significantly greater than the increase rate of memory speed
* Therefore, the time to access memory is a bottleneck for the processor performance
* Cache memory:
  + on the first level of memory hierarchy, small dimension, fast access (compared to main mem) and it has the function of decreasing the mean time to acces memory.
  + Cache Hit
    - CPU requests data that are available in the cache. Hit Rate is the percentage of “cache hits” over the total of data accesses.
  + Cache Miss
    - CPU requests data that is not in the cache. The fail time corresponds to the time required to transfer data to cache. Dependent on machine arquitecture
  + Cache L1
    - in the same pachage as the processor
    - misses are faster to solve than L2 misses
  + Cache L2
    - installed in a separated package
  + Spatial location
    - when a data element is requested their neighbors will also be
    - a cache line is read in a single operation
    - it is efficient to request data elements from the same cache line
  + Temporal location
    - when a data element is requested it has a high probability of being requested again in a short period of time
    - the user should guarantee that the data that is in the cache is used with more frequency

**Processor architecture**

* Cache
* Parallelism
  + parallel functional units (hyperthreading)
* pipelining

**Cache coherence**

* Multicore processors
  + cores share a common address space
  + caches are independent per core
* A memory system is coherent if
  + a read after a write of a location X from the same processor P must return the last write
  + a read by P after a write by Q must return the last write, if both instructions are sufficiently separated in time and no other write occurs by any processor
  + writes to the same location are serialized
    - two writes from different processors are seen in the same order by all processors
* Write-invalidate protocol
  + to enforce coherence
    - ensure that a single processor has exclusive access to a memory location it wants to write to
    - the exclusive access invalidates all copies that may exist in other processors
    - other processors that had a copy of the memory location are forced to read the value again
  + most protocols use blocks of data instead of single memory locations (can lead to false sharing)

**Shared-memory model VS Message-passing model**

* Shared-memory model
  + number of active threads is 1 at start and finish of program, changes dynamically during execution
  + execute and profile sequential program
  + incrementally make it parallel
  + stop when further effort not warranted
* Message-passing model
  + all processes active throughout execution of program
  + sequential-to-parallel transformation required major effort
  + transformation done in one giant step rather than many tiny steps

**Incremental Parallellization**

* Sequential programming is a special case of a shared-memory parallel program
* Parallel shared-memory programs may only have a single parallel loop
* Incremental parallelization: process of converting a sequential program to a parallel program a little bit a time

**Execution Context**

* every thread has its own execution context
* execution context: address space containing all of the variables a thread may access
* contents of execution context:
  + static variables
  + dynamically allocated data structures in the heap
  + variables on the run-time stack
  + additional run-time stack for functions invoked by the thread

**Shared and private variables**

* shared variable:
  + has the same address in execution context of every thread
* private variable:
  + has different address in execution context of every thread
* a thread cannot access the private variables of another thread

**Clauses**

* clause:
  + an optional, additional compoennt to a pragma
* private clause
  + directs compiler to make on or more variables private
* firstprivate clause:
  + private variables are undefined on thread entry
  + “firstprivate” is used to create private variables having initial values identical to the variable controlled by the master thread as the loop is entered
* lastprivate clause
  + sequentially last iteration:
    - iteration that occurs last when the loop is executed sequentially
  + used to copy back to the master thread’s copy of a variable, the private copy of the variable from the thread that executed the sequentially last iteration
* nowait clause
  + compiler puts a barrier synchronization at the end of every parallel for statement
* schedule clause
  + schedule type required, chunk size optional
  + allowable schedule types:
    - static: static allocation
    - dynamic: dynamic allocation
    - guided: guided self-scheduling
    - runtime: type chosen at run-time based on value of environment variable OMP\_SCHEDULE
  + scheduling options: (ex. shared mem ppts, slide 52)
    - schedule(static): block allocation of about n/t contiguous iterations to each thread
    - schedule(static, C): interleaved allocation of chunks of size C to threads
    - schedule(dynamic): dynamic one-at-a-time allocation of iterations to threads
    - schedule(dynamic, C): dynamic allocation of C iterations at a time to threads
    - schedule(guided, C): dynamic allocation of chunks to tasks using guided self-scheduling heuristic. Initial chunks are bigger, later chunks are smaller, minimum chunk size is C
    - schedule(guided): guided self-scheduling with minimum chunk size 1
    - schedule(runtime): schedule chosen at run-time based on value of OMP\_SCHEDULE;
      * Unix example: setenv OMP\_SCHEDULE “static, 1”

**Parallel pragma**

The “parallel” pragma precedes a block of code that should be execute by all of the threads

Note: execution is replicated among all threads

**Performance Improvement**

* Too many fork/joins can lower performance
  + inverting loops may help performance if
    - parallelism is in inner loop
    - after inversion, the outer loop can be made parallel
  + or, by defining outside the parallel region
* maximize parallel regions
  + reduce number of fork/joins
* if loop has too few iterations, fork/join overhead is greater than time savings from parallel execution
* the “if” clause instructs compiler to insert code that determines at run-time whether loop should be executed in parallel
  + ex. “#pragma omp parallel for if (n > 5000)”
* #pragma omp for loop – has an implicit barrier
* we can use “schedule” clause to specify how iterations of a loop should be allocated to threads
* static schedule:
  + all iterations allocated to threads before iterations executed
* dynamic schedule:
  + only some iterations allocated to threads at beginning of loop’s execution. Remaining iterations allocated to threads that complete their assigned iterations.

**Static vs Dynamic Scheduling**

* Static
  + low overhead
  + may exhibiy high workload imbalance
* Dynamic
  + higher overhead
  + can reduce workload imbalance

**Chunks**

* contiguous range of iterations
* increasing chunk size reduces overhead and may increase cache hit rate
* decreasing chunk size allows finer balancing of workloads

**parallel sections Pragma**

* precedes a block of k blocks of code that may be executed concurrently by k threads
* syntax: “#pragma omp parallel sections”

**section Pragma**

* precedes each block of code within the encompassing block preceded by the parallel sections pragma
* may be omitted for first parallel section after the parallel sections pragma
* syntax: “#pragma omp section”

**sections pragma**

* appears inside a parallel block of code
* has same meaning as the parallel sections pragma
* if multiple sections pragmas inside one parallel block, may reduce fork/join costs

**task Construct**

* binding
  + thread set -> inner most parallel region (current parallel team)
* variables
  + by default variables are firstprivate. To share we need to say it explicitly
* no need to create extra tasks
  + child tasks are executed concurrently with their parent
* taskwait
  + specifies a wait on the completion of child tasks generated since the beginning of the current task
  + when a thread encounters a taskwait construct, the current task is suspended untill all child tasks that it generated before the taskwait region complete execution.
* taskgroup
  + when a thread encounters a taskgroup construct, it commences to execute the taskgroup region
  + at the end of the taskgroup region, the current task is suspended until all child tasks that it generated in the taskgroup region and all of their descendant tasks complete execution
* taskloop
  + specifies that the iterations of one or more associated loops will be executed in parallel using explicit tasks
  + the iterations are distributed across tasks generated by the construct and scheduled to be executed

**Main difference between parallel regions and tasks**

* once a parallel region is created:
  + no threads in the team can leave the region until the end of the region
  + no threads can join the parallel region

**OpenMP vs MPI**

A picture containing text, screenshot, font, number

Description automatically generated

**Speedup and Scalability:**

* ideal is to get the work done P times faster on P processors
* strong scaling: compute a fized-size problem P times faster
  + linear speedup occurs when S=P
  + Amdahl Law limits it. No Speedup for serial part
* weak scaling: compute a problem P times bigger in the same amount of time
  + Speedup depends on the amount of serial work remaining constant or increasing slowly as the size of the problem grows
  + assumes amount of communication among processors also remains constant or grows slowly

**Performance measures**

Performance measures are always given for a combination of an algorithm and a machine

The parallelization of an algorithm may have 2 objectives:

* reduction of the processing time by using more processing units -> strong scaling
* allow computation of a higher dimension problem on more processors -> weak scaling

**Speedup**

* speedup: ration between sequential processing time and parallel processing time
* several measures of Speedup:
  + relative: Tseq is obtained by executing the parallel code in a single node of the parallel machine
  + real: Tseq is obtained by executing the most efficient sequential code in a single node of the parallel machine
* observed speedup: obtained by measuring the program execution
* analytic speedup: obtained by time complexity analyses (computational model)
* Speedup Analyses:
  + difficult to measure if a single processor dooes not have enough memory to execute the program when high dimension problems are solved
  + for the real and relative speedup, the use of lower capacity processors as well as less efficient code, results in higher speedup values
    - this is why the parallel computation time should be given with the speedup obtained
* scaled speedup – weak scaling
  + considering that in a single computer it is not possible to measure the sequential time due to time and memory limitations

A black text on a white background

Description automatically generated with low confidence**Efficiency**

* measures the utilization rate of the processors in the execution of a parallel program. It is given by the ratio of Speedup and the number of processors used
* the efficiency has a value between 0 and 1, and it is a measure of the parallelization quality

**Scalability Analysis**

* To evaluate the adaptability of the algorithm-machine to solve higher dimension problems

**A system is scalable is the efficiency can be ept constant (between 0 and 1) when P and n are inscreased**

**Scalability measures**

* iso-efficiency function
  + represents the growth rate of n in relation to P that keeps the Efficiency constant
  + a small growth rate of n in relation to P (ex. linear): system with a high scalability factor
  + a high growth rate of n in realtion to P (ex. exponential): system with a low scalability factor. It means that we need to increase n in relation to P in order to keep Efficiency constant
* iso-granularity curves
  + for systems with low scalability factors, the computation of the Iso-efficiency function is not feasible due to the high computation times
  + alternatively, the Iso-granularity curves show the machine computation capacity when the work (granularity) is kept constant by processor

**Scalable algorithms**

* an efficient and scalable algorithm typically has the following characteristics:
  + the work can be separated into numerous tasks that proceed almost totally independently of one another
  + communication between the tasks is infrequent or unnecessary
  + lots of computation takes place before messaging or I/O occurs
  + there is little or no need for tasks to communicate globally
  + initiate as many tasks as possible