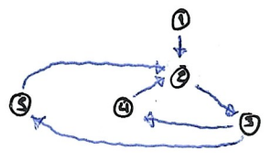
how to program multicore processors - compilers don't do the job and even for sequential programming we need to write code carefully if we want to get performance and scalable programs. **Main challenge** - write scalable programs that keep the efficiency level as data increases and as more cores are available.

main goal of parallel computing - scalable (resource-aware) computing. **Resources** - 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 analysis

modern scientific method:



1- problem
2- observation
3- theory
4- experimentation
5- numeric simulation

use parallel computing - possibility of solving bigger problems and with more realistic representation, reduce the development costs, have higher freedom to explore alternatives

sustained performance - depends on several factors (I/O speed, data access pattern, memory hierarchy, algorithm design), the relevant performance is the one that results from the real execution of an algorithm

amdahl law:

s is sequential
(1-s) is parallelized
p - number of processors
T_p - total execution time
s - sequential time susceptible of parallelization

$$Speedup = \frac{T_1}{T_p} = \frac{T_{seq}}{T_{par}} \qquad T_p = \frac{(1-s)}{p} + s$$

$$Speedup = \frac{1}{\frac{1-s}{p} + s} \qquad Speedup = \frac{1}{s} \Rightarrow p \to \infty$$

Δ — program execution time

- allows to have a realistic expectation, for a given algorithm, about what we can obtain with the parallel approach
- shows that to achieve higher speedups it is necessary to reduce or eliminate the algorithm sequential blocks

gives a comparison metric to measure the scalability of several algorithms for the same problem

functional parallelism - independent tasks execute different operations on different data sets

data parallelism - independent tasks execute the same operation over different data

streaming - to process streams of data divide the process in steps and the number of steps limits the speedup. Ex: process multiple streams of data, real time data analysis, real time decision making support.

shared memory model - each processor (a core) executes a thread, threads interact by shared variables, number of forks/joins influences performance, each thread has its own process state, but share global variables defined by the master thread

race condition - undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time. It occurs when two or more threads can modify the same memory location at the same time

critical section - portion of code that only a thread at a time may execute, we denote a critical section by putting the pragma omp critical in front of a block of C code

false sharing - occurs when 2 or more threads access different data on the same cache line (read/write). Ex example: access close positions of a global vector

distributed memory model - parallel program = a set of tasks executing concurrently. Tasks interact by sending messages through the communication channels.

Methodology:

problem → 1 → 2 → 3 → 4

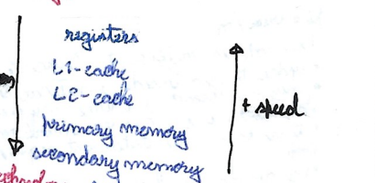
1- partitioning
2- communication
3- agglomeration
4- mapping

mapreduce - model for writing applications that can process big data in parallel on multiple nodes. Provides analytical capabilities for analyzing huge volumes of complex data

sequential operations - operations that require some effort to be parallelized

parallel operations - operations that are embarrassingly parallel

memory hierarchy:



memory technology - faster memories → are more expensive per bit because they requires more area per bit

cache memory - is on the first level of the memory hierarchy, small dimension, fast access (compared to main memory), and it has the function of decreasing the mean time to access memory

execution time = clock cycles executing user code + clock cycles for data transfer between cache and main memory

cache hit: the 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: the CPU requests data that is not in the cache. The fail time corresponds to the time required to transfer data to cache. It is dependent on the machine architecture.

cache L1 and L2 - L1 cache is on the same package as the processor and L2 cache is installed in a separated package, L1 misses are faster to solve than L2 misses.

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 high probability to be requested in a short period of time, the user should guaranty that the data that is in the cache is used with more frequency.

multicore processors - cores shares a common address space, caches are independent per core

cache coherence - **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.

to enforce coherence:
- ensure that a single processor has exclusive access to a memory location it wants to write
- the exclusive access invalidates all copies that may exist in others processors
- other processors that had a copy of that 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)

extract parallelism

OpenMP - API for parallel programming on multicores, sufficient to program multicores and easy to convert sequential data parallel programs

fork - master thread creates or awakens additional threads to execute parallel code

join - at end of parallel code created threads die or are suspended

parallel programming models

message-passing model - all processes active throughout execution of program, sequential to parallel transformation requires major effort, transformation done in one giant step rather than many tiny steps

incremental parallelization - process of converting a sequential program to a parallel program a little bit at a time

pragma - compiler directive in C or C++, stands for "pragmatic information", a way for the programmer to communicate with the compiler, compiler free to ignore pragmas

execution context - address space containing all of the variables a thread may access

contents: 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

function omp_get_num_procs - returns number of physical processors available for use by the parallel program

private clause - directs compiler to make one or more variables private

function omp_set_num_threads - uses the parameter value to set the number of threads to be active in parallel sections of code, may be called at multiple points in a program.

declaring private variables - either loop could be executed in parallel, we prefer to make outer loop parallel to reduce the number of forks/joins, we then must give each thread it's own private copy of variable j

firstprivate clause - used to create private variables having initial values identical to the variable controlled by the master thread as the loop is entered

lastprivate clause - 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

reduction - are so common that OpenMP provides support for them, may add reduction clause to parallel for pragma, specify reduction operation and reduction variable, OpenMP takes care of storing partial results in private variables and combining partial results after the loop

nowait clause - compiler puts a barrier synchronization at end of every parallel for statement

parallel pragma - precedes a block of code that should be executed by all of the threads, execution is replicated among all threads

performance improvement:
- inverting loops or, if one outside the parallel region
- maximize parallel regions
- conditional parallelism
- optimize barrier use
- use schedule clause to specify how iterations of a loop should be allocated to threads

static schedule - all iterations allocated to threads before any iterations executed, low overhead, may exhibit high workload imbalance

dynamic schedule - only some iterations allocated to threads at beginning of loop's execution, remaining iterations allocated to threads that complete their assigned iterations, higher overhead, can reduce workload imbalance

chunk - contiguous range of iterations

scheduling options: **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)** guided self-scheduling with minimum chunk size 1, **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 (runntime)** schedule chosen at run-time based on value of OMP_SCHEDULE

functional parallelism - to this point all of our focus has been on exploiting data parallelism, OpenMP allows us to assign different threads to different portions of code

task version - any thread can work on any task resulting in a more efficient thread usage, there is only one parallel region, and therefore the user can control better the number of threads used

scaled speedup - obtain a more realistic speedup measure, considering that in a single computer it is not possible to measure the sequential time due to time and memory limitations. Weak scaling.

$$scaled\ speedup = \frac{T_s + P \times T_p'}{T_s + T_p'}$$

T_p' - is the parallel processing time for P processors of the parallel component

efficiency - measure of the parallelization quality

$$E = \frac{speedup}{P}$$

scalability analysis - to evaluate the adaptability of the algorithm - machine to solve higher dimension problems

characteristics of an efficient and scalable algorithm:
- 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

distributed system - collection of distinct processes which are spatially separated and which communicate with one another by exchanging messages. If system is distributed, if the message transmission delay is not negligible compared to the time between events in a single process.

message based communication

Notes:

- fork/join parallelism only keeps master thread active
- shared-memory model only keeps 1 active thread at start and finish of program, changes dynamically during execution
- shared-memory model execute and profile sequential program, incrementally make it parallel and stop when further effort not warranted
- sequential programming is a special case of a shared-memory parallel program
- parallel shared-memory programs may only have a single parallel loop
- openMP makes it easy to indicate when the iterations of a loop may execute in parallel
- compiler takes care of generating code that forks/joins threads and allocates the iterations to threads
- every thread has its own execution context
- shared variable has 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
- clause is an optional, additional component to a pragma
- private variables are undefined on thread entry
- sequentially last iteration is an iteration that occurs last when the loop is executed sequentially
- increasing chunk size reduces overhead and may increase cache hit rate, decreasing chunk size allows finer balancing of workloads
- strong scaling: compute a fixed-size problem P times faster
- weak scaling: compute a problem P times bigger in the same amount of time

```
double area, pi, x;
int i, n;
...
area = 0.0;
# pragma omp parallel for private(x)
for (i=0; i<n; i++) {
    x = (i+0.5)/n;
# pragma omp critical
    area += 4.0/(1.0+x*x);
}
pi = area/n
```

This code is inefficient. Sources:

- update to area inside a critical section
- only one thread at a time may execute the statement; i.e., it is sequential code
- time to execute statement significant part of loop
- by amdahl's law we know speedup will be severely constrained