Parallel Computing

Introduction

"Free lunch" and Moore's Law

Moore's Law was the name of a phenomenon that the performance of sequential computers was observed to increase exponentially, with a doubling rate of 18-24 months. Although this isn't what Gordon Moore speculated, the "law" held from the 70s up until the early- to mid-2000s. This increase in performance made building and selling parallel computers more difficult, which caused many companies in this field to fold.

Given that this phenomenon is now over due to physical restrictions on processors, parallel computing is here to save the day.

Processor Performance

Performance is usually measured in the maximum, best-case number of operations (FLOPS) that can be carried out per unit of time (seconds). This can be problematic, given that the nominal performance of a processor depends on two factors:

- If the used algorithm contains the right operations, dependencies etc. to allow full utilisation of the processor (solving a graph problem usually executes 0 FLOPS)
- If the memory system is able to supply the necessary data for keeping the processor busy

The second point is important, given that memory access time did not keep pace with processor performance, which led to the introduction of ever larger and more caches in processors.

Parallel vs Distributed

What is the difference between parallel and distributed computing?

Whereas the focus of parallel computing is to use parallel resources (such as processors) efficiently for solving given **computational** problems (algorithms), distributed computing is concerned with making independent, non-dedicated resources available and cooperate with each other.

PRAM Model

Models are useful tools for designing and analysing parallel algorithms and there are multiple models in parallel computing with no single one of them being able to *bridge* all of the parallel computer architectures together. The most useful of those is the **PRAM** model, a generalisation of the *Random Access Machine* used in sequential computing.

The PRAM assumes a large memory where processorsynchroniseds can read and write words in unit time. The number of processors can be chosen as needed (fixed, or as a function of the input size) and they all execute their own program, but do so in strictly synchronised lock-step. Therefore, the machine is always in a well-defined state. Due to many processors operating in lock-step, concurrent access to memory will happen. There are three variations how to deal with situations like these:

- **EREW** - *Exclusive Read Exclusive Write* - No access to memory for more than one processor at the same time.
- **CREW** - *Concurrent Read Exclusive Write* - Simultaneous reads by more than one processor are possible
- **CRCW** - *Concurrent Read Concurrent Write* - Simultaneous reads and writes are allowed, if two or more processors try writing to the same memory region it has to be ensured that the same value is written. Another way would be a **priority CRCW PRAM** which has some priority associated with its processors and only one will successfully write its value into the memory.

In terms of computability, **EREW** is the **weakest** of these models, since there are problems which can be solved in $O(1)$ by a **CRCW** but in $O(\log n)$ by an **EREW**. Conversely, **every EREW program runs on a CRCW PRAM but not the other way around**.

```
par (0<=i<n) b[i] = 1; // a[i] could be

par (0<=i<n, 0<=j<n) {
  if (a[i]<a[j]) b[i] = 0; // a[i] is not
}

par (0<=i<n) if (b[i]) x = a[i];
```

Algo 1: Finding the maximum of n numbers stored in an array, runs in $O(1)$ parallel time steps, using n^2 processors on a CRCW PRAM

	Found	*Processors*	Operations
Common CREW	$O(1)$	n^2	$O(n^2)$
CRCW PRAM	$O(\log n)$	$\frac{n}{2}$	$O(n)$

```
par (0<=i<n, 0<=j<m) { C[i,j] = 0;
for (k=0; k<l; k++) {
    C[i,j] += A[i,k]*B[k,j];
  }
}
```

Algo 2: n x n matrix multiplication in $O(n)$ time steps and $O(n^3)$ operations on CREW PRAM

Flynn's taxonomy

Different characterisation of parallel machines, looks at the instruction and data streams of the system.

- **SISD** - *Single Instruction, Single Data* - Sequential Computer
- **SIMD** - *Single Instruction, Multiple Data* - A single instruction can operate on a larger batch of data, e.g. a vector (array)
- **MIMD** - *Multiple Instruction, Multiple Data* - General PRAM, each processor has own instructions and own data
- **MISD** - *Multiple Instruction, Single Data* - Deeply pipelined system, single data stream passes through different stages
- **SPMD** - *Single Program, Multiple Data* - All processors execute the same program, but each processor can be in a different part of said program

Sequential and Parallel Time

Parallel algorithms are, in general, compared to parallel the **best possible** sequential algorithm for solving a given problem. In case this algorithm is not known, the **best known** algorithm is used instead. The reasoning is simple, we want to improve the best possible sequential solutions by using parallelism.

The running times, in number of steps taken, for sequential and parallel algorithms on *worst-case inputs (often impossible and therefore "typical" inputs instead) of size n and p* are $T_{seq}(n)$ and $T_{par}(p, n)$ respectively. The time for the parallel algorithm is measured as the time for the last processor-core to finish, with the assumption that all cores started at the same time.

Absolute speed-up is the relation between the best known sequential algorithm and the parallel algorithm for inputs of size $O(n)$.

$$S_p(n) = \frac{T_{seq}(n)}{T_{par}(p, n)}$$

`Absolute Speedup`

Example:

We have a parallel algorithm $T_{par}(p, n) = O(\frac{n}{p})$ and a best known sequential algorithm $T_{seq}(n) = O(n)$. **In this case the absolute speedup would be $O(p)$.**

Such a speedup, which is independent of n is also called **linear** or **perfect**, which is very rare and the best possible speedup. The argument here is that the parallel algorithm is used to construct an even faster sequential algorithm.

Given that linear speedup is the best possible outcome, we know that the best possible parallel algorithm runs for at least as long as the sequential algorithm divided by the cores.

$$T_{par}(p, n) \geq \frac{T_{seq}(n)}{p}$$

Cost and Work

The cost of a parallel algorithm is defined as $p * T_{par}(p, n)$ which denotes for how much time the system is occupied. The term **work** denotes the number of operations which are carried out by the algorithm ("*work is time*"). For a parallel algorithm the work is the total work carried out by the p cores, excluding time and operations spent idling by some processors. Therefore, **work is operations carried out by assigned processors**.

If the cost $p * T_{par}(p,n) = O(T_{seq}(n))$ then we have a cost-optimal parallel algorithm. **Cost-optimal algorithms have linear speedup!**

If the work $W_{par}(p,n) = O(T_{seq}(n))$ then we have a work-optimal parallel algorithm.

A work-optimal algorithm which is not cost-optimal can have linear speedup for a smaller range of processors. An example would be a parallel algorithm executing a sequential algorithm on one of p processors. This is a work-optimal parallel algorithm, but definitely not cost-optimal.

The absolute speedup measures how well a parallel algorithm is outperforming a best known sequential algorithm, but is not measuring if the algorithm *itself* is able to exploit the p processors well. This notion, also called *scalability*, is expressed by the relative speedup

$$SRel_p(n) = \frac{T_{par}(1,n)}{T_{par}(p,n)}$$

`Relative Speedup`

If we assume that we have an arbitrary large number of processors available, any parallel algorithm can achieve the fastest running time denoted by $T\infty(n) = T_{par}(p',n)$. Therefore, any parallel algorithm runs for at least the same amount of time or longer, meaning that $T_{par}(p,n) \geq T\infty(n)$.

$$\frac{T_{par}(1,n)}{T\infty(n)}$$

`Parallelism`

The *parallelism* of an algorithm is both the **largest achievable speedup** as well as the **largest number of processors** for which linear, relative speedup can be achieved. Choosing a larger number of processors leads to a speedup which is inferior to the linear one.

Therefore the differences between absolute and relative speedup are clear:

- **Relative Speedup** - Compare algorithm to itself, did I fully utilise all processor cores?
- **Absolute Speedup** - Is my parallel algorithm better than the best known sequential one?

We also know that the running time of a sequential algorithm is less than or equal to a parallel algorithm executed on a single core, therefore we know that the absolute speedup is at most only as large as the relative speedup.

$$S_p(n) \leq SRel_p(n)$$

Summarising:

Relative	Absolute	Linear
$SRel_p(n) = \frac{T_{par}(1,n)}{T_{par}(p,n)}$	$S_p(n) = \frac{T_{seq}(n)}{T_{par}(p,n)}$	$T_{par}(p,n) \geq \frac{T_{seq}(n)}{p}$
Compare par alg against itself	Compare par alg against best sequential	Best possible speedup
`How well are` p `exploited?`	`How well improved from sequential?`	

Overhead

Overhead is an inevitable cost of synchronising the workload of the parallel algorithm. We differentiate between *finely grained* (frequent communication) and *coarsely grained* (rare communication) computation.

Load imbalance happens when the work amongst the different processor cores is divided up unevenly, which leads to a lesser speedup.

$$T_{par}(i,n) - T_{par}(j,n)$$

`Load imbalance`

A good load balance means that $T_{par}(i,n) \approx T_{par}(j,n)$ for all processors. Achieving this is called *load balancing* wherein we differentiate between static load balancing (work is divided upfront) and *dynamic load balancing* (work is divided according to the communication during the execution). Problems where the input and work can be statically distributed to the processors, and no further explicit interaction is required are called **embarrassingly, trivially or pleasantly parallel**.

The division of work between processors can either be **oblivious**, meaning only input size is taken into consideration, or **adaptive**, meaning the input itself is taken into consideration.

Amdahl's Law

Amdahl's law states that if you assume that you can split the work performed by a sequential algorithm into a strictly sequential part s, which is independent of n, and a perfectly parallelisable part $r = (1 - s)$ than we know that the **maximum speedup that can be achieved by `Par` over `Seq` is** $\frac{1}{s}$ (Proof S16, Script)

This leads to the realisation that any algorithm with a sequential, non-parallelisable part will limit and kill the possible speedup. This means that such an algorithm can't be used as the basis of a good, parallel algorithm.

Usual victims of this law are I/O-operations, long chains of dependent operations, sequential data structures, etc.

in a good parallel algorithm the sequential part will not be a constant fraction but depend on the input and even decrease with it. In such a case Amdahl's Law doesn't apply and a good speedup can be achieved with large enough inputs.

Efficiency

The parallel running time of an algorithm with good, linear speedup can be written as

$$T_{par}(p,n) = O\left(\frac{T(n)}{p + t(n)}\right)$$

Whereas $T(n)$ is the parallelisable term and $t(n) = T\infty(n)$ non-parallelisable.

Therefore, *scaled speedup* can be calculated as follows:

$$S_p(n) = \frac{T_{seq}(n)}{T_{par}(p,n)} = O\left(\frac{T(n)}{\frac{T(n)}{p+t(n)}}\right) = O\left(\frac{1}{\frac{1}{p} + \frac{t(n)}{T(n)}}\right) \rightarrow O(p)$$

Which shows that, the faster $t(n)$ and $T(n)$ converge, the faster the speedup converges. Hence, parallel efficiency which is calculated by comparing `Par` to the best possible parallelisation of `Seq` is defined as:

$$E_p(n) = \frac{T_{seq}(n)}{p * T_{par}(p,n)} = \frac{S_p(n)}{p}$$

With the following properties:

- $E_p(n) \leq 1$
- If $E_p = e$ (some constant) $\Rightarrow$ **`linear speedup`**
- Cost-optimal algorithms have a constant E_p

Weak scalability

`Par` is weakly scaling relative to `Seq` if by keeping the average work w per processor $\frac{T_{seq}(n)}{p}$ constant, the running time $T_{par}(p,n)$ remains constant. The input size scaling function is $g(p) = T_{seq}^{-1}(p * w)$.

The iso-efficiency function $f(p)$ tells how n should grow as a function of p to maintain constant efficiency, and should not grow faster than the input size scaling function $g(p)$, which tells how much n can at most grow if the parallel time is to be kept constant. Therefore:

$$f(p) = O(g(p))$$

Said differently: if an algorithm does not have constant efficiency and its speedup is independent of n, then calculate n as function of p.

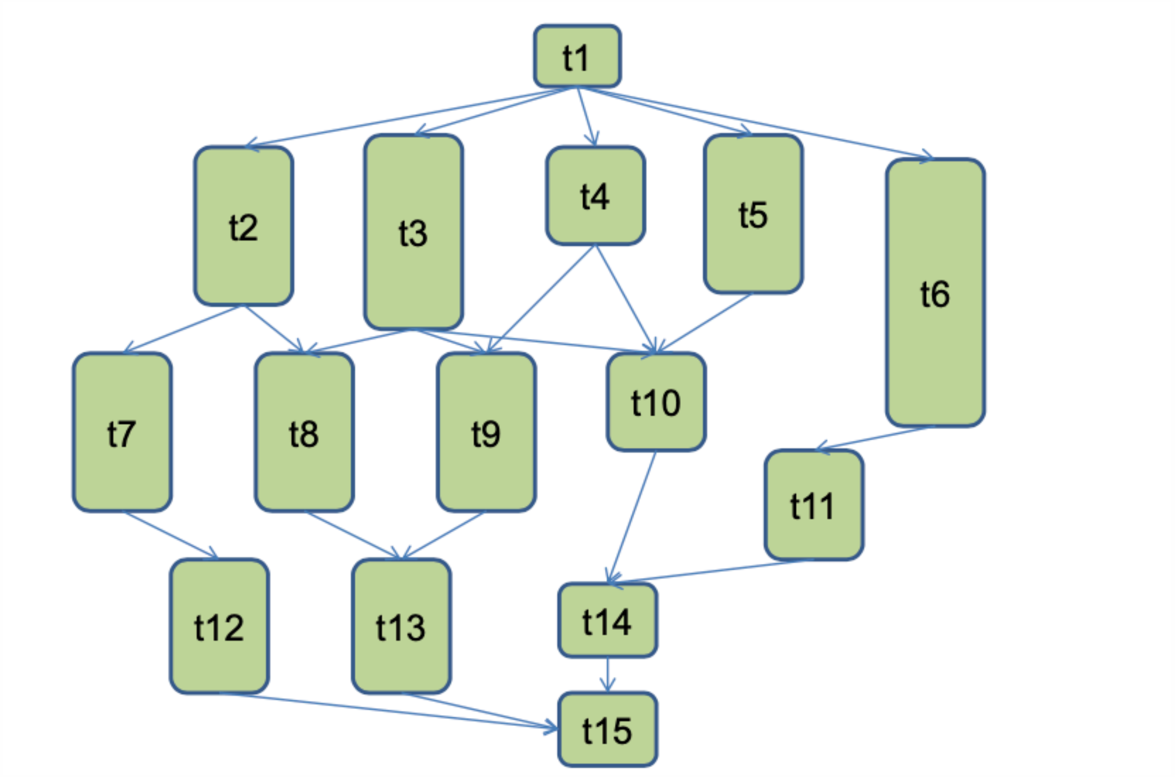
To actually analyse the performance of a parallel algorithm against its sequential counterpart we have two tools at hand:

- **Strong scaling analysis** - Keep n constant, if the parallel time decreases proportionally to p the algorithm is strongly scalable
- **Weak scaling analysis** - Keep the average work per processor constant by increasing n, if the parallel running time remains constant the algorithm is weakly scalable

DAG Graphs

Computation can be structured as a collection of tasks $t_1, t_2, \ldots t_n$ where a task t_i is a sequential computation (also called *strand*). Such a task needs input data from other tasks, and produces output data for other tasks. Accordingly, a task cannot be executed before said data is available.

These **dependencies** are captured by a DAG $G = \langle V, E \rangle$ where $V = \{t_1, t_2 \ldots t_n\}$ and E are the data dependencies. Task t_j is dependent on task t_i if there is a path from t_i to t_j. From this follows that independent tasks can be executed in parallel.



After executing a node all outgoing edges are removed, which produces a new, smaller DAG. Then execute all ready (= no incoming edges) tasks in this DAG and continue until done.

The **work per task** is denoted as W_i for t_i. Therefore for the work, it follows that:

$$W(n) = \sum_{0 \le i < k} W_i(n) \ge T_{seq}(n)$$

On p processors, the best possible execution takes at least $\frac{W(n)}{p}$ steps. Given a p processor schedule $T_p(n)$ we now know that the following must be true:

$$T_p(n) \ge \frac{W(n)}{p} \ge \frac{T_{seq}}{p}$$

No matter how we schedule, the amount of work can never be completed faster than the best possible parallelisation ($T\infty(n)$) since the number of operations in the most operation-intensive path from input to output notes dictates the final speed. Therefore:

$$T\infty(n) = \sum \text{heaviest Path: } W_i(n)$$

The work on such a heaviest path is also called the *span*, or the *depth* of a given graph G. Such a graph is also referred to as *critical path* with length $T\infty$. From these observations two laws can be defined:

- **Work Law** - $T_p(n) \geq \frac{W(n)}{p} \geq \frac{T_{seq}}{p}$, a p processor schedule is at least as large as the work performed per processor
- **Depth Law** - $T_p(n) \geq T\infty(n)$, a p processor schedule is at least as large as the fastest possible parallelisation (since the heaviest path dominates)

Furthermore, the maximum speedup in a DAG is defined as $\frac{W(n)}{T\infty(n)}$, it also gives the largest number of processor-cores for which linear speedup could be possible.

One possible scheduling algorithm is simple greedy scheduling, which is considered to be *two-optimal*. Having a best possible p processor schedule $T_p^*(n)$ it holds that:

$$T_p(n) \leq 2T_p^*(n)$$

Independence of Program Fragments

There are three so called *Bernstein conditions* to determine the kind of dependency between two program fragments P_i and P_j. Each of these programs has a set of input variables I_i and a set of output variables O_i. The two program fragments are dependent if either one of these cases is true:

- $O_i \cap I_j \neq \varnothing$ - true or flow dependency
- $I_i \cap O_j \neq \varnothing$ - anti-dependency
- $O_i \cap O_j \neq \varnothing$ - output-dependency

Inside a loop there are equivalent loop dependencies:

In a *loop carried flow dependency* the outcome of an earlier iteration affects the computation of a later iteration. Such iterations cannot be done in parallel.

```
// loop carried flow dependency
for (i = k; i < n; i++) {
  a[i] = a[i] + a[i-k];
}
```

In a *loop carried anti-dependency* the outcome of a later iteration would have affected an earlier iteration, if the two iterations were reversed or carried out simultaneously

```
// loop carried anti-dependency
for (i = 0; i < n-k; i++) {
  a[i] = a[i] + a[i+k];
}
```

In a *loop carried output dependency* two iterations write to the same output variables, which would make the output undefined if executed simultaneously (race conditions).

```
// loop carried output dependency
for (i=0; i<n-k; i++) {
  a[0] = a[i];
}
```

Merging ordered sequences

Given two ordered sequences A and B with n and m elements, respectively, the standard sequential algorithm for merging is straight-forward. The problem with this algorithm is, that it is strictly sequential since the output for a position depends on the relative order of all previous elements. The complexity of such a standard algorithm is $O(n + m)$.

To be able to parallelise this merging operation one possible solution is to use **merging by ranking**. The approach goes as follows:

For each element $A[i]$ in A find the position j in B such that $B[j] < A[i] < B[j + 1]$, this position is called the **rank of $A[i]$ in B**. Knowing the rank of the element $A[i]$ in B we also know the position of $A[i]$ in the final output array C which is $i + rank(A[i], B)$. This approach can be performed in parallel in $O(\log max(n, m))$ time steps, which is not work-optimal given that the work is the same as the sequential merging by ranking algorithm.

To turn this into a work-optimal algorithm the elements aren't ranked separately but in disjoint, consecutive blocks roughly the size of $\frac{n}{p}$. (For details see the script, I don't really get it)

Master theorem

Many recursive algorithms have a specific form of recurrence relation. To solve this kind of relation without having to go for induction every time, there is a way using the so called master theorem (for simple, regular divide-and-conquer recurrences):

Given a recurrence of the form

$$T(n) = a * T\left(\frac{n}{b}\right) + \Theta(n^d \log^e n)$$

with the constants being

$$a \geq 1, \quad b > 1, \quad d \geq 0, \quad e \geq 0, \quad and \quad T(1) \text{ some constant}$$

the recurrence has the following solution cases:

1. $T(n) = \Theta(n^d \log^e n)$ if $\frac{a}{b^d} < 1$ (or equivalently $\frac{b^d}{a} > 1$)
2. $T(n) = \Theta(n^d \log^{e+1} n)$ if $\frac{a}{b^d} = 1$ (or equivalently $\frac{b^d}{a} = 1$)
3. $T(n) = \Theta(n^{\log_b a})$ if $\frac{a}{b^d} > 1$ (or equivalently $\frac{b^d}{a} < 1$)

Furthermore, the different constants are different factors of the recursive procedure, with b being the **shrinkage** factor by which the subproblems get smaller, and a the **proliferation** or **expansion** factor, denoting roughly the *number of subproblems* to be solved at each recursion level, with the number of levels being $\lceil \log_b n \rceil$.

Shared-Memory Parallel Systems and OpenMP

Caches and locality

The first difference between a **real** shared-memory system, and a naive model is the existence of caches. Given that improvements in memory performance have not kept up with improvements in processor performance, the ratio of access times between data values fetched from memory and data in cache has increased over time.

The usual cache system of a processor does not work on a level of single values, but instead is using blocks of memory addresses. Such blocks, usually 64 bytes in size, can be mapped to a **cache line**. Therefore a cache line stores a memory block (and some metadata).

There are different mapping strategies: A cache in which each memory block is mapped to a predetermined cache line is called a **directly mapped** cache whereas a **fully associative** cache is a cache where each memory block can be mapped to any cache line. Modern browsers use **set associative** caches with small k-set sizes ($k = 2, 4, 8, \ldots$) where each memory block can be mapped to some predetermined set. Therefore, a directly mapped cache is in fact nothing but a *1-way set associative cache*.

When reading *words*, a processor calculates the memory block to which the word belongs. If it is in the cache we are talking about a *cache hit*, otherwise about a *cache miss*. On a *miss*, a new block needs to be read into a corresponding cache line, with some strategy needed to empty a full cache in case that there is no space left.

There are three types of cache misses:

- **Compulsory (cold) miss** - no blocks are in the cache, therefore every first reference will lead to a cache miss
- **Capacity miss** - the cache is full, some cache lines need to be evicted
- **Conflict miss** - All cache lines in the set are occupied, which can happen even if the cache as a whole is not full

Writing to memory involves the cache as well in different ways:

- **Write allocate** - If the block of the address written is already in the cache, it must be overwritten (otherwise a subsequent read could deliver outdated results)
  - typically employing the **write back** strategy, which keeps the data in the cache until its evicted and only then written into the memory
- **Write no-allocate** - If the block of the address is not yet in the cache, a cache line for that block needs to be allocated or the address updated directly in memory
  - typically employing the **write through** strategy, which immediately passes on each write to the main memory

Caches can be crucial for the runtime of an algorithm, with the order of the three loops for matrix-matrix multiplication (see slides, script and exercises) being responsible for a difference of a factor of about 20-40 between the best and worst loop orders, all down to cache miss rates.

Multi-core caches

Usually modern multi-processor systems have a hierarchy of caches of increasing size (L1, L2, L3, ...) with L1 being usually the smallest at the lowest level, closest to the processor-core. In parallel multi-core systems several new problems arise with this system.

The first one is **cache coherence** among private caches, which is about the question of what happens to a memory block which is in the private cache of two cores, but updated in one. If the cache line will *eventually* be updated in the other core, too we are talking about a *coherent* cache system, otherwise it is *non-coherent*. Keeping all caches coherent requires an algorithm which is called a *cache coherence protocol* which can potentially affect performance due to excessive cache coherence traffic.

A *cache coherent* system needs to fulfil the following three criteria:

- **Local consistency** - If core c writes to a at a time t_1 and reads a at a later time t_2, with no other writes between t_1 and t_2 then c reads the value written at t_1.
- **Update transfer** - If core c_1 writes to a at time t_1 and another core c_2 reads a at a later time t_2, with no other writes between t_1 and t_2 then c_2 reads the value written by c_1 at t_1.
- **Write consistency** - If cores c_1 and c_2 write to a at the same time, then one of the two values is stored at a.

The second problem is **false sharing** which is caused by the granularity of the cache system. Given that cache lines store entire blocks of memory it could happen that two cores are operating on different data in the **same cache line**, which are, therefore, completely independent operations. Due to the fact that the memory locations are in the same block, each update of a value will cause *cache coherence activity* although it is completely unnecessary, leading to a significant performance loss.

Modern memory systems are very *NUMA*, meaning every processor has its own "local" RAM but every other core has direct access via a collective address space (*distributed shared memory*). This gives rise to the possibility of a super-linear speedup, which is easily explained:

Assume we have an algorithm which can be parallelised well in the sense that the working set with p processors is $\frac{1}{p}$ of the working set on just one processor. If p keeps growing, the ever decreasing working set will fit in ever faster caches of the memory hierarchy, leading to a situation in which the parallel algorithm is much faster than the sequential equivalent.

Application performance

Therefore the nominal performance of the CPU does not alone determine the performance of a given application on a system. If a memory system is not able to supply data fast enough to the processor-cores, the performance of the memory system will determine the performance.

An application can be either *memory-bound* or *compute-bound*:

- **Memory-bound** - the operations to be performed per unit read from or written to the memory take **less** time than reading/writing from/to memory. Therefore, memory access times will determine the application's performance.
- **Compute-bound** - the operations to be performed per unit read from or written to the memory take **more** time than reading/writing from/to memory. Therefore, nominal processor performance will determine the application's performance.

Memory consistency

When a program is executing sequentially, all reads and writes appear to take place in the execution order of the program's instructions *(program order)*. Similarly, when two or more programs are being executed concurrently, a natural expectation would be that the outcome of the execution will be as if some *interleaving* of the two programs is taking place. This particular kind of memory consistency is called *sequential consistency*. Unfortunately, due to the existence of *per-core write buffers*, modern multi-core systems usually are **not** sequentially consistent. Write buffers are special buffers used to reduce the time for writing into main memory by buffering the write operations and executing these operations in batches at some point in time.

An illustrative example can be found in the script on page 49, but can be summarised as follows:

If we have a program segment which should be executed at most by one core and is protected by flags, this setup would only hold under the assumption of sequential consistency. If both cores update their respective flags, these updates end up in their respective write buffers, meaning one core does not know that the program segment has already been accessed by the other core. The result is an outcome of write and read instructions which did not follow program order.

pthreads

A thread is an independent stream of instructions that can be scheduled by the OS. It is usually the smallest unit that can be independently scheduled by the OS. Every such thread has local stack variabls, registers and *thread local storage*.

The programming model can be summarised as follows:

- **Fork-join type parallelism** - A thread can spawn any number of new threads and wait for threads to terminate. These threads are referenced by an identifier
- Initially one **master thread** is active. All threads execute within the same program (SPMD) but can be different functions
- Spawned threads are "peers", any thread can wait for the termination of any other thread
- Threads are scheduled by the underlying system and *may or may not* run simultaneously
- There is **no implicit synchronisation between threads**, they progress independently and asynchronously
- Threads share process global data (since a thread always belongs to a process)
- Coordination mechanisms exist for protecting access to shared variables (locks, condition variables). All concurrent updates must be protected to prevent undefined outcomes.

Runnable threads are expected to be scheduled to free cores. This is the assumption of the *SMP* (Symmetric Multi-Processor) which assumes a single OS, all cores to be treated equally and threads not to be bound to cores. There can easily be more threads than cores. Given that threads can be assigned freely, cache problems can arise if for example a thread loads data into a cache, goes to sleep and is moved to another core which does not have the data in its cache, leading to a cache miss.

Obviously, this programming model can easily lead to race conditions which need to be prevented by known synchronisation constructs (semaphores, locks, monitors, ...). A special kind of race condition is the so called *data race* which is a situation where two or more threads access a shared object, and at least one of these accesses is a write operation. Determining if a program will have data races is an undecidable problem, so it is impossible to algorithmically find *all* race conditions.

A function is **thread-safe** if it can be executed concurrently by any number of threads and always produce correct results. Typical **non-thread-safe** functions are:

- Functions that do not protect shared variables (side effects)
- Functions that keep state over successive invocations (static variables)
- Functions that return pointers to static variables

Synchronisation between threads by using locks can easily lead to deadlocks and other undesirable behaviour. An alternative is a **conditional wait** which makes it possible that a thread which is currently in a critical section, but can't continue, can be suspended and relinquish its acquired lock. Upon waking up the suspended thread it receives the lock again and can continue with its computation. Another useful tool are **atomic operations** which are complex operations that are ensured to be executed atomically.

An algorithm is **lock-free** if several threads are attempting to execute the operation and at least one of the threads can make progress. On the other hand an algorithm is **wait-free** if a thread is attempting to execute an operation and is guaranteed to be able to make progress, regardless of other threads. If an algorithm is wait-free it is also lock-free, not vice versa.

OpenMP

OpenMP is another fork-join thread model with threads being less explicit than in pthreads. Again, a master threads can fork a consecutively numbered set of working threads which together share executing work specified by a work sharing construct (e.g. loop of independent iterations). Upon completion, threads are joined again. An OpenMP program is a single program with all forked threads executing the same code (SPMD).

OpenMP's main characteristics are:

- **Implicit parallelism** through work sharing. All threads execute the same program.
- **Fork-join parallelism** - master thread implicitly spawns threads through parallel region construct, threads join at the end of this parallel region
- **Unique identifier** for each thread in such a parallel region, consecutively numbered
- The number of threads can exceed the number of processors
- Constructs for sharing work across threads.
- Threads can share variables, which are shared among all threads. Threads can also have private variables.
- Unprotected, parallel updates of shared variables lead to **data races**
- A correct OpenMP program is **always a correct sequential program!** Meaning if the compiler does not recognise `#pragma` commands it should still execute normally.

```
#include <omp.h>

#pragma omp parallel [clauses]
...
  parallel region
...
```

OpenMP also has a number of library calls which allows the programmer to look up the number of threads, the current thread's id, etc.

```
int omp_get_thread_num(void);
int omp_get_num_threads(void);
int omp_get_max_threads(void);
void omp_set_num_threads(int num_threads);
```

Work sharing ideas

The work done by the algorithm can be split in different ways. The most common way is the for loop which splits chunks of the loop-iteration and distributes it to other threads

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i<n; i++) {
    // code
  }
}
```

Loops can only be parallelised if the number of iterations is known beforehand. Furthermore, `break`s are illegal, whereas `continue`s are okay. Loops need to be in *canonical form*, meaning that the increments have to be in one of the forms among `i++, i+=inc, or i=i+inc` and upper bounds `i<n, i<=n, i>n, i>=n, or i!=n`.

Another way is the usage of **tasks**, which are code sections that can be suspended and later be executed by any thread in the parallel region. Tasks are often used with recursive calls such as in the `Quicksort` algorithm.

```
void QuickSort(int x[],int n) {
  if (n<=1) return;
  pivot = choosepivot(x,n);
  ix = partition(x,n,pivot);

  #pragma omp task shared(x)
  QuickSort(x,ix);
  #pragma omp task shared(x)
  QuickSort(x+ix+1,n-1-ix);
}
```

The third work sharing construct are **sections**, which are used rarely these days. The idea is that after identifying specific blocks of code which can be executed in parallel we can put these explicitly in sections which are then parallelised.

```
#pragma omp parallel
{
  #pragma omp sections
  #pragma omp section
  {
    // block A
  }

  #pragma omp section
  {
    // block B
  }
  ...
}
// block A and B are independent of each other and can be executed in parallel
```

If a region is to be executed by only one thread there are two ways to do this, either only the `master` thread can execute the block of code or any thread is allowed to.

```
#pragma omp single [clauses] // any thread
#pragma omp master // only master
// no mutual exclusion!
```

An important note regarding the use of these constructs is the **implicit barrier**. At the end of a `single` construct OpenMP has an implicit barrier to synchronise threads again, which can be avoided by the `nowait` clause. The `master` construct on the other hand does not have an implicit barrier and needs an explicit `#pragma omp barrier`. **Important:** if not all threads can reach the barrier it will result in a deadlock.

Per default, all variables declared before a parallel region are shared by all threads in said region. On the other hand, variables declared inside the parallel region are *local* to each thread and therefore private. The sharing of such variables can be controlled by sharing clauses in the `#pragma` directive

```
private(<comma separated list of variables>) // variables declared here not
initialized!
firstprivate(<comma separated list of variables>) // vars initialized to value before
par region
shared(<comma separated list of variables>)
default(shared|none) // sharing vars declared by master thread before parallel region
(or not)
```

Important clauses

Loop schedules

OpenMP also provides so called **loop schedules** which dictate how large chunks of a loop are and how they will be distributed amongst all threads.

`schedule(<type>, [, <chunksize>])`

- `static` - Iterations are divided into chunks of size `chunksize` (optional, default is approximately $\frac{n}{p}$)
- `dynamic` - Chunks are distributed to threads depending on their availability (default `chunksize` is 1)
- `guided` - Like dynamic, but the actual chunksize for each thread is calculated as $\frac{\text{unassigned iterations}}{p}$ with the `chunksize` parameter being the smallest possible value (default again 1)

Collapse

```
#pragma opm parallel for collapse(2)
for (i=0; i<n; i++) {
  for (j=0; j<n, j++) {
    x[i][j] = f(i,j);
  }
}
```

Nested loops can be parallelised, with the loops then handled as one with an iteration space of `n*m` iterations. Collapsing is only possible if the iteration spaces are known prior to the outer loop, in other words **`m` cannot depend on `n`**.

Reduction

OpenMP provides a special `reduction` clause for performing an operation on an array across all threads which accepts an operator and a variable in which the end result is to be stored. OpenMP gives no guarantee for the performance of such reductions.

```
sum = 0;

#pragma omp parallel for reduction(+:sum)
for (i=k, i<n; i++) {
  sum = sum + a[i];
}
```

Critical

Mutual exclusion can be enforced with the use of named critical sections which are statically designated at compile time. These sections can be executed by at most one thread at a time. Shared variables can be updated in a critical section since such updates are not race conditions.

```
#pragma omp critical [(name)]
```

Example of a data race being resolved with a critical section:

```
int t = 0;
#pragma omp parallel shared(t) {
  t++;
  print(“Threads so far %d of %d by %d\n”,
t,omp_get_num_threads(),omp_get_thread_num());
  // data race on shared variable, undefined value since on thread reads while another
writes
}
```

This read / write conflict can be solved with an explicit barrier

```
int t = 0;
#pragma omp parallel shared(t) {
  t++;
  #pragma omp barrier
  print(“Threads so far %d of %d by %d\n”,
t,omp_get_num_threads(),omp_get_thread_num());
// still a data race on t++ operation, undefined update result
}
```

Lo and behold, the critical section saves the day

```
int t = 0;
#pragma omp parallel shared(t) {
  #pragma omp critical // mutual exclusion in critical section
  t++;
  #pragma omp barrier
  print(“Threads so far %d of %d by %d\n”,
t,omp_get_num_threads(),omp_get_thread_num());
}
```

Entering a critical section is **costly**, so one should employ the strategy of checking and rechecking (after entering the critical section). If check fails, an unnecessary entry into the critical section is saved.

Distributed memory parallel systems and MPI

Communication algorithms in networks

Distributed memory systems feature an interconnection network which is needed for communication between connected processors. The *topology* (structure) of such a network, both direct and indirect, can be modelled as a (un)directed, unweighted graph $G = \langle V, E \rangle$ where V are the processors and E the connection between the.

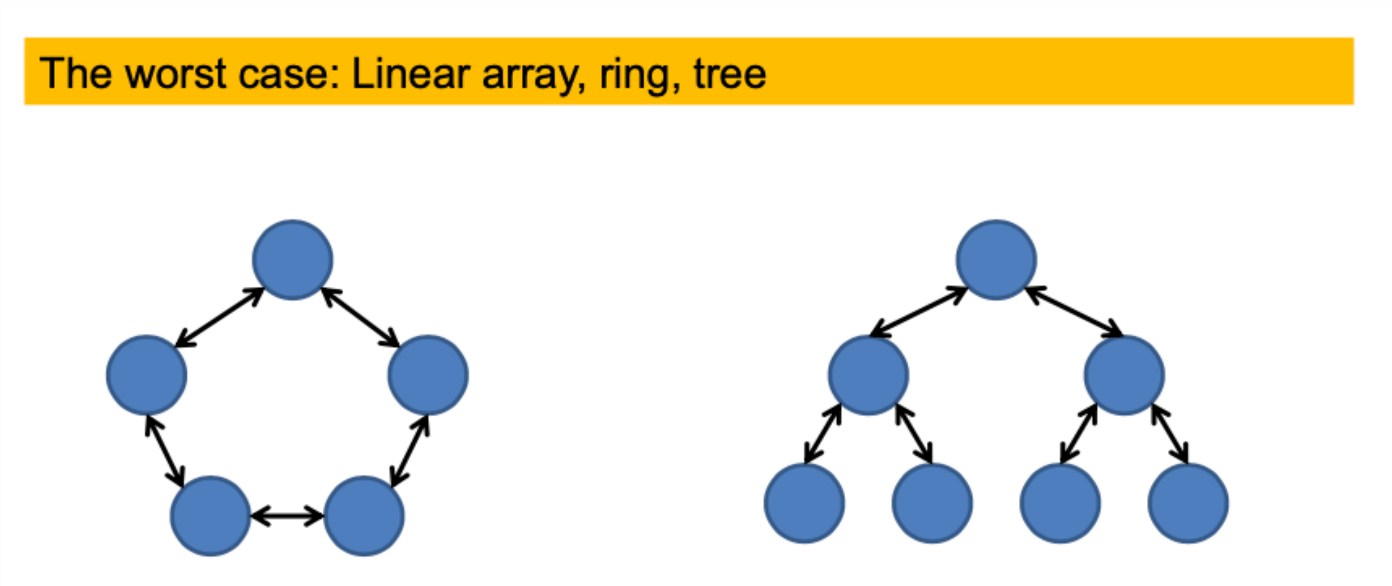
Communication can be *unidirectional* (one direction) or *bidirectional* (two-way street, most modern systems). The diameter of a graph is defined as the longest path between a pair of nodes, or formally:

$$diam(G) = max\{dist(u, v) \mid u, v \in V\}$$

The degree of a graph on the other hand is the largest number of outgoing edges from a node in G

$$degree(G) = max\{degree(u) \mid u \in V\}$$

The *bisection width* of a graph is the smallest number of edges that must be removed, in order for the graph to be split into two approximately equally large parts. Finding $bisec(G)$ is essentially the NP-complete $\textsc{GraphPartitioning}$ problem. Tree networks however have $bisec(T) = 1$. The worst possible communication networks which still support parallel computing are the *linear processor array* and the *processor ring*.



The main problem in these networks is the **broadcast problem** which is about how to transmit data from the root node to every other node in the minimum needed communication steps.

The lower bound is $\lceil \log_{k+1} p \rceil$ in a k-degree network with p nodes. The broadcast problem for an arbitrary graph is again NP-complete.

Communication costs are calculated considering units of data of size m with a fixed, startup latency α and a transmission time per unit β - therefore the linear transmission cost model states that transmitting m units from processor u to v takes

$$\alpha + \beta m$$

time units. This is just a crude approximation of the cost of communication between processors in a network.

MPI

The idea of the message-passing interface is to structure parallel computations as sequential processes with no shared information. These processes communicate explicitly by sending and receiving *messages* between each other.

MPI's main characteristics are:

- Processes are ordered by rank in communication domains
- Ranks are successive $0, 1, \dots p-1$ with p the number of processes in the communication domain
- Multiple such communication domains are possible, making the isolation of certain tasks possible
- Processes operate on local data and all communication is explicit
- Communication is reliable, ordered and network oblivious (communication between **all** processes is possible)

A typical MPI program is initialised by calling `MPI_Init` with the typical `argc` and `argv` variables and is finished with a call to `MPI_Finalize(void)`.

Communicators

After the MPI library has been initialised the processes are put into a communication domain called `MPI_COMM_WORLD`, these communication domains are called *communicators* in MPI which are distributed objects that can be operated upon by all belonging processes. Such a communicator is referenced by a *handle* of type `MPI_Comm`. Processes can look up the size (number of processes) and each process can determine its own rank in a communicator.

```
// usually after the MPI_Init call
int rank, size;
MPI_Comm_size(MPI_COMM_WORLD,&size);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

All communication is relative to a communicator, meaning that processes from different communicators cannot communicate with each other. This allows the construction of safe, parallel libraries since communication in different communicators can never interfere. It is also possible to create new communicators with several MPI library functions, such as `MPI_Comm_dup`, `MPI_Comm_split`, etc which allows creating new communicators from existing ones.

Point to point communication

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,int source, int tag, MPI_Comm
comm, MPI_Status *status);
```

Sending and receiving are both explicit processes in MPI. By posting a `MPI_Send` call, a sending process initiates and completes sending data to a receiving process. By posting the equivalent `MPI_Recv` call, a receiving process declares itself ready to receive up to the described amount of data. The call completes when the data sent has been received. For communication to take place, both processes must give the same *message tag* to the message, and the sending process needs to specify the receiving process' rank. The space for MPI data is almost exclusively in the user space, therefore it is important to always allocate enough buffer space for the data which is being sent and received.

Point to point communication is **deadlock free** provided that the destination posts a matching receive operation. It is **non-overtaking**, meaning messages sent arrive in the sent order at the destination (can be changed using tags). It is also **reliable**, messages are not lost or manipulated in transition.

`MPI_Send` has non-local completion which means it may block for a receive operation that could never happen (deadlock).

Given that `MPI_Recv` can specify more elements than actually sent in the corresponding call, we need some functionality to find out how much data was actually sent. This information is available in the `status` object and can be obtained by calling one of these two functions

```
int MPI_Get_count(const MPI_Status *status, MPI_Datatype datatype, int *count);
int MPI_Get_elements(const MPI_Status *status, MPI_Datatype datatype, int *count);
```

In a ring-like organisation of processes, the following implementation will lead to a communication deadlock, since each process is waiting to receive data from the previous process, which will never be sent because every process is waiting to receive the data.

```
#define TAG 1000

MPI_Status status;
int *a = ...;
int *b = ...;

MPI_Recv(a, count, MPI_INT, (rank-1+size)%size, TAG, comm, &status);
MPI_Send(b, count, MPI_INT, (rank+1)%size, TAG, comm);
```

Such a situation can be solved with a `MPI_Sendrecv` function which combines the functionality of a blocking send and a blocking receive operation. Thus, a process can at the same time, concurrently, send and receive data to and from two other processes in the communicator without the risk of deadlocks.

The code above can be resolved like so

```
#define TAG 1000

MPI_Status status;
int *a = ...;
int *b = ...;
MPI_Sendrecv(b, count, MPI_INT, (rank+1)%size, TAG, a, count, MPI_INT, (rank-
1+size)%size, TAG, comm, &status);
```

Communication can be **determinate** or **non-determinate** depending on the parameters in the `MPI_Recv` call. For example, specifying a wildcard `MPI_ANY_SOURCE` a process can receive a message from any other process, which is therefore non-determinate. Likewise, the tag argument can be wild-carded by using `MPI_ANY_TAG`. Careless programming can lead to multiple problems and undefined behaviour here.

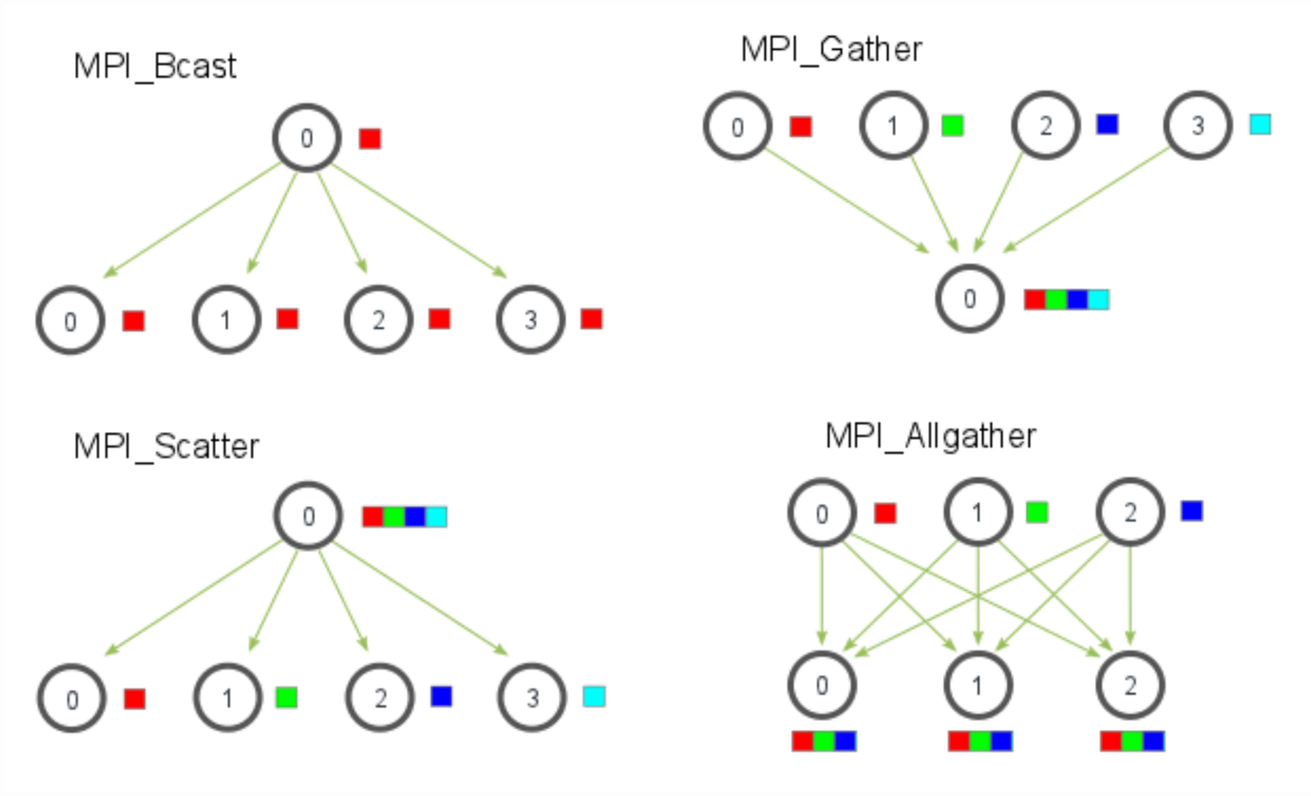
One-sided communication

An alternative to the point to point communication where two processes are explicitly involved, there is one-sided communication where one process alone is explicitly initiating the communication, and therefore has to specify what is happening on both sides. MPI provide operations for retrieving data `MPI_Get`, transferring data `MPI_Put` amongst other different functions. All one-sided communication calls are non-blocking.

Since processes do not share address spaces in any way, a different mechanism is needed to expose parts of the memory for interaction with another process. This data structure is called *window* which has a handle of type `MPI_Win`. Several `MPI_Win_` operations are provided such as `MPI_Win_create` which is a collective operation for creating a communication window in which each process gives the process local address and size of the memory it will expose. The memory which is supposed to be exposed needs to be allocated in advance, either by using standard `malloc()`-like operations or with MPI provided operations. Using stack allocated data in a communication window is dangerous since the memory could go away before the window is freed.

Collective operations

Many MPI operations are so-called **collective operations** which means that it has to be called by every single process in the communicator. If one process issues such a collective operation then all other processes must also eventually call the operation with no other collective call before.


MPI_Bcast
MPI_Gather
MPI_Scatter
MPI_Allgather