CP Course Notes

José Duarte

June 23, 2020

1 Parallel Architectures

1.1 Flynn's Taxonomy

1.1.1 Instruction Level

- Single Instruction (SI) - System in which all processors execute the same instruction
- Multiple Instruction (MI) - System in which different processors execute may execute different instructions

1.1.2 Data Level

- Single Data (SD) - System in which all processors operate on the same data
- Multiple Data (MD) - System in which different processors may operate on different data

1.1.3 Existing Architectures

	SD	MD
SI	SISD	SIMD
MI	MISD	MIMD

Figure 1: Possible processor architectures. MISD is the exception.

SISD The classic Von Neumann architecture, present in serial processors

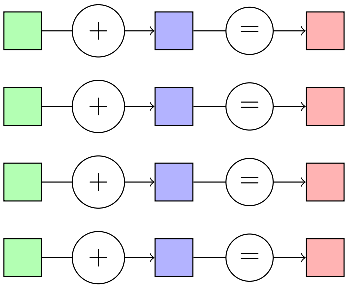

Figure 2: SISD Processing Example. One instruction is applied to each data item.

SIMD The data is divided among the processors and each item is subjected to the same sequence of instructions. Present in modern CPUs through Advanced Vector Extensions (AVX), as well as GPUs.

MISD Possible to envision as a pipeline at most, since it is not possible for multiple instructions to execute at the same time over the same data item.

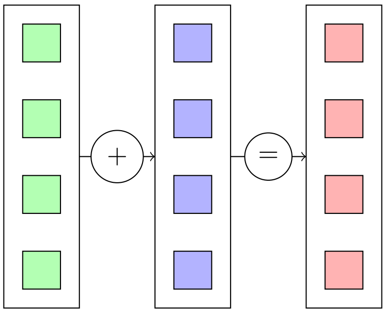

Figure 3: SIMD Processing Example. One instruction is applied to a block of data items.

MIMD Collection of autonomous processors executing independent programs, such as a distributed network or cluster. MIMD architectures can be categorized under four designations, each with different purposes, some of these architectures may be composed to form more complex systems.

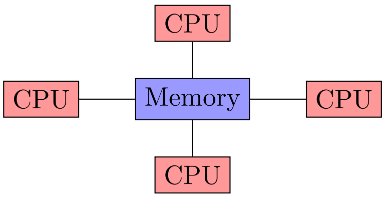

Figure 4: Uniform Memory Access (UMA). This architecture suffers from the need of synchronization among memory accesses. We can picture this as several CPUs accessing common RAM.

1.2 Software Taxonomies

Applying Flynn's taxonomy to software we observe the following patterns.

- Data Parallel (SIMD) - Parallelism that is a result of identical operations applied concurrently on different data items. This approach is difficult to apply to complex problems.
- Single Program, Multiple Data (SPMD) - A single program is run across multiple processing units (computers, processors, threads). Processing units execute the same code but do not work in lock-step.

1.3 Memory

Memory can be split into shared memory or distributed memory.

1.3.1 Shared Memory

Shared memory represents a global memory space, it is accessed by every processor. Each processor has a local cache (L1, L2, L3) which keep portions of the memory for faster access. The L1 and L2 caches are available at a core level while the L3 cache is shared among cores.

Shared memory is easier to program with, several frameworks are available (e.g. OpenMP, OpenACC). Low latency for data sharing between tasks. However the programmer is responsible for synchronization, furthermore the memory access is non-uniform (NUMA) and accesses to global memory (RAM) can be a bottleneck.

1.3.2 Distributed Memory

Memory is shared between processes using messages (OpenMPI), this allows to exploit distributed computing architectures in a cost effective way.

Memory scales based on the number of processors and the access to local memory is fast.

This approach however is complex due to difficulties in programming communication effectively and data structures may be difficult to distribute.

2 Parallel Performance

2.1 Performance

In computing performance can be defined by two factors, computational requirements or resources. Computational requirements can be thought of "what needs to be done?", or efficacy, and computational resources can be thought in terms of "how much will it cost?", or efficiency.

$$Performace \sim \frac{1}{Resources\ for\ solution} \tag{1}$$

2.1.1 Expectations

In a scenario where we have p processors and each processor is rated at f MFLOP, should we see $f \times p$ MFLOPS performance?

The answer is not as simple as "yes". Several causes may affect performance, while causes may interact with each other, they need to be understood separately.

2.1.2 Embarrassingly Parallel Computations

Or for short EPCs, are computations that can be trivially divided into several independent parts able to be executed simultaneously. For *truly* EPCs there should be no interaction between processes, while in *nearly* EPCs the input and output is required to be distributed and combined in some way.

EPCs have potential to achieve maximal speedups in parallel platforms.

2.1.3 Latency & Throughput

Latency The time it takes to complete a task is called latency. The scale can be anywhere from the nanoseconds to days, lower latency is better. It is related to *response time.*

Throughput The rate at which a series of tasks can be completed is called throughput. It has units of work per time unit, a larger throughput is better.

2.1.4 Speedup, Efficiency & Scalability

Speedup and efficiency are important metrics for parallelism relating to performance.

Speedup Speedup compares the latency for solving the identical computational problem on one hardware unit versus on P hardware units.

$$S_p = \frac{T_1}{T_p} \tag{2}$$

Where T_1 is the latency of the program with one worker and T_P is the latency on P workers.

Efficiency It is the speedup divided by the number of workers.

$$E_p = \frac{S_p}{p} \tag{3}$$

It measures the return on hardware investment. The ideal efficiency is 1, corresponding to a linear speedup.

Cost A parallel algorithm is cost-optimal if $C_p = T_1$ or equivalently $E_p = 1$.

$$C_p = p \times T_p \tag{4}$$

Scalability As previously discussed the performance of a parallel solution is subjected to several factors, namely scalability. Scalability is the ability of a parallel algorithm of achieving performance gains proportional to the number of processors and the size of the problem.

2.1.5 Asymptotic Complexity

Asymptotic complexity is the key to comparing algorithms. Comparing absolute times is not particularly meaningful since they are specific to the hardware.

Time Complexity An algorithm's time complexity summarizes how the execution time of algorithm grows with the input size.

Space Complexity Similarly to time complexity, space complexity summarizes how the required amount of memory grows with input size.

Big O Notation Denotes a set of functions with an upper bound. $O(f(N))$ is the set of all functions $g(N)$ such that there exist positive constants c and N_0 with $|g(N)| \leq c \cdot f(N)$ for $N \geq N_0$.

Big Ω Notation Denotes a set of functions with a lower bound. $\Omega(f(N))$ is the set of all functions $g(N)$ such that there exist constants c and N_0 with $g(N) \geq c \cdot f(N)$ for $N \geq N_0$.

Big Θ Notation Denotes a set of functions with both upper and lower bounds. $\Theta(f(N))$ means the set of all functions $g(N)$ such that there exist positive constants c_1, c_2 and N_0 with $c_1 \cdot f(N) \leq g(N) \leq c_2 \cdot f(N)$ for $N \geq N_0$.

2.2 Amdahl's Law

Amdahl's law fixes the problem size and varies the number of processors, hence the denomination of Fixed Size Speedup. The law relates at the reduction of the execution time. Let f be the sequential fraction of a program, $1 - f$ is the part that can be parallelized.

$$\begin{aligned} S_p &\leq \frac{T_1}{T_p} = \frac{T_1}{(f \cdot T_1) + \frac{(1_f)T_1}{p}} \\ S_p &\leq \frac{1}{f + \frac{1-f}{p}} \\ S_{p \rightsquigarrow \infty} &\leq \frac{1}{f} \end{aligned} \tag{5}$$

Scalability When considering scalability, Amdahl's law refers to the abillity of a parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem.

Application Amdahl's law is applicable under the following circumstances:

- When the problem size is fixed.
- Strong scaling ($p \to \infty$, $S_p = S_\infty \to \frac{1}{f}$).
- Speedup bound is determined by the degree of sequential execution time in the computation.

Example If 90% of the computation can be parallelized, what is the maximum speedup achievable using 8 processors?

We first calculate the sequential part of the program:

$$\begin{aligned} 0.9 &= 1 - f \\ f &= 0.1 \end{aligned}$$

And then apply Amdahl's law:

$$\begin{aligned} S_8 &\leq \frac{1}{0.1 + \frac{1-0.1}{8}} \\ S_8 &\leq 4.7 \end{aligned}$$

2.3 Gustafson-Barsis' Law

Also denominated as Scaled Speedup, it is interested in larger problems when scaling. In other words, for the same time, how much more work can we do?

Considering a the non-parallelizable part and b as the parallelizable part, we calculate T_P, for P processors, as follows:

$$\begin{aligned} T_P &= a + P \cdot b \\ T_1 &= a + 1 \cdot b \\ &= a + b \end{aligned} \tag{6}$$

Given that the wall-clock execution time is always the same, the scaled speedup is calculated on the volume of data processed.

$$\begin{aligned} S_P &\leq \frac{T_P}{T_1} \\ &\leq \frac{a + P \cdot b}{a + b} \end{aligned} \tag{7}$$

Consider $\alpha = \frac{a}{a+b}$ as the sequential fraction of the parallel execution time. We can then define the scaled speedup as follows:

$$\begin{aligned} S_P &\leq \alpha + P \cdot (1 - \alpha) \\ &\leq P - \alpha \cdot (P - 1) \end{aligned} \tag{8}$$

Scalability The ability of a parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem.

Application The Gustafson's law applies under the following scenarios:

- When the problem size can increase.
- When the number of processors increases.
- Speedup function include the number of processors.
- Can maintain or increase parallel efficiency as the problem scales.

Example An application executing on 64 processors spends 5% of the total time on non-parallelizable computations. What is the scaled speedup?

$$\begin{aligned} S_64 &\leq P - \alpha \cdot (P - 1) \\ &\leq 64 - 0.05 \cdot (64 - 1) \\ &\leq 60.85 \end{aligned} \tag{9}$$

2.4 Computational DAG

A parallel program maybe be represented as a directed acyclic graph (DAG), this graph has several properties, such as the span, which can be used to evaluate the potential performance of an algorithm.

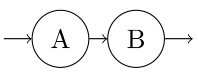

Figure 5: Serial Composition

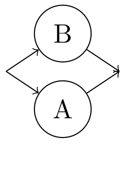

Figure 6: Parallel Composition

Work The total number of tasks of the graph is called *work*, this is equivalent to running the algorithm with a single processing unit (T_1).

$$T_p \geq \frac{T_1}{P} \tag{10}$$

The work is equal whether considering T_1 or T_P, that is:

$$T_1(A \cup B) = T_1(A) + T_1(B) \tag{11}$$

Span The span, or the critical-path length, is the minimum of sequential steps the algorithm must execute.

$$T_p \geq T_\infty \tag{12}$$

The span is different when comparing T_1 and T_P.

$$\begin{aligned} T_\infty(A \cup B) &= T_\infty(A) + T_\infty(B) \\ T_\infty(A \cup B) &= max\{T_\infty(A), T_\infty(B)\} \end{aligned} \tag{13}$$

2.4.1 Speedup

The speedup on P processors is given by:

$$\frac{T_1}{T_P} \tag{14}$$

If $\frac{T_1}{T_P} = P$ we have *linear* speedup. When $\frac{T_1}{T_P} \geq P$ we have *superlinear* speedup, this however is not possible due to the work law (Equation 10).

2.4.2 Parallelism

Parallelism can be defined as the average amount of work per step along the span, that is:

$$\frac{T_1}{T_\infty} \tag{15}$$

2.4.3 Work-Span Model

Considers a greedy scheduler, that is, no worker is idle while there are tasks to execute.

- T_P - running time with P workers.
- T_1 - running time with 1 worker, serial execution.
- T_∞ - the time taken along the critical path.

The critical path is the sequence of tasks that has the longest execution time.

Lower-Bound A lower-bound can be calculated with the following formula:

$$max\left(\frac{T_1}{P}, T_\infty\right) \leq T_P \tag{16}$$

Upper-Bound

$$T_P \leq \frac{T_1 - T_\infty}{P + T_\infty} \tag{17}$$

3 Parallel Programming

Parallel programs often start as sequential programs. This approach has the advantage of having a previous codebase to validate the new approach, furthermore it is easier to start with a sequential approach given the ease of programming and debugging.

To parallelize a program we first have to evaluate if it is worth the effort, to do so we first identify the hotspots with a profiler.

The parallelization process starts with the hotspots, applying small changes with testing in-between.

3.1 Finding Concurrency

The first step in parallelizing a candidate program is to find concurrency, in other words, we have to find tasks that can be executed at the same time.

To do so, we need to consider flexibility, efficiency and simplicity.

3.1.1 Task Decomposition Guidelines

Flexibility The program design should afford flexibility in the number and size of the tasks generated.

Efficiency Tasks should have enough work as to amortize the cost of creating and managing them. furthermore, they should be sufficiently independent so that managing dependencies does not become the bottleneck.

Simplicity The code must remain as simple and readable as possible so as to be debuggable and quick to understand and modify.

3.1.2 Data Decomposition Guidelines

Often implied by task decomposition, data decomposition is a good starting point when the main computation is organized around manipulation of large data structures or similar operations are applied to different parts of the data structure.

Flexibility The size and number of data chunks should support a wide range of executions.

Efficiency Data chunks should generate considerable amounts of work to minimize the communication and management impact.

Simplicity Complex data compositions can get difficult to manage and debug.

3.2 Algorithmic Structure Design Space

The high level approach of an algorithm can be split in to three possibilities, all approaches can then be split into two sub approaches, the linear or recursive approach.

- Organize by Tasks - the linear approach to this problem is to make use of task parallelism, the recursive approach is through divide and conquer.

- Organize by Data Decomposition - the linear approach of data decomposition is by making use of geometric decomposition (arrays), the recursive approach is through recursive data structures.
- Organize by Data Flow - the linear solution is to make use of the pipeline pattern, the recursive approach is based on event coordination.

3.3 Implementation Environment

3.3.1 Program Structures

Single Program Multiple Data In the SMPD approach all tasks execute the same program in parallel but each has its own set of data.

Master/Worker Pattern The master/worker is works by having a thread distribute work amongst the others, the workers execute concurrently and each worker takes tasks from the bag of tasks. It is suited for embarrassingly parallel problems.

Loop Parallelism Pattern This pattern exploits the iterative nature of some programs by parallelizing the execution of loop iterations. This requires the loop not to have dependencies among iterations.

Fork/Join Pattern A main task forks off some number of other tasks that then continue in parallel, the main task then waits for completion before moving on.

Pipeline Pattern Tasks are applied in sequence to data.

4 Parallel Patterns

A parallel pattern is a recurring combination of task distribution and data access that solves a specific design problem in parallel algorithm design.

4.1 Nesting Pattern

Nesting is the ability to hierarchically compose patterns. This pattern appears in both serial and parallel algorithms.

It is a compositional pattern, allowing other patterns to be composed in such way that task blocks can be replaced with another pattern.

4.2 Control Patterns

4.2.1 Serial Control Patterns

Structured serial programming is based on following patterns.

Sequence Ordered list of tasks that are executed in a specific order.

Selection Condition c is first evaluated. Either task a or b is executed depending on the true or false result of c.

Iteration Condition c is evaluated, if it is true a is evaluated again, afterwards c is evaluated again. This repeats until c is false.

Recursion Dynamic form of nesting allowing functions to call themselves.

4.2.2 Parallel Control Patterns

Parallel control patterns extend serial control patterns. Each parallel control pattern is related to at least one serial control pattern.

Fork-Join Allows control flow to fork into multiple parallel flows, then rejoin later.

Map Performs a function over every element of a collection, it replicates a serial iteration pattern.

Stencil Elemental function accesses a set of "neighbors", stencil is a generalization of map.

Reduction Combines every element of a collection using an associative "combiner function".

Scan Computes all partial reductions of a collection. For every output in a collection, a reduction of the input up to that point is computed.

Recurrence More complex version of map, where loop iterations can depend on one another.

4.3 Data Management Patterns

4.3.1 Serial Data Management Patterns

Random Read & Write Memory locations indexed with addresses, used with pointers. Aliasing can cause problems when parallelizing the code.

Stack Allocation Useful for dynamically allocate data in LIFO manner. It preserves locality and when parallelized, since each threads gets its own stack thread locality is preserved.

Heap Allocation Useful when data cannot be allocated in LIFO manner. Slower and more complex when compared with stack allocations. A parallelized allocator should be used when allocating memory in parallel.

Objects Objects are language constructs to associate data with code to manipulate and manage that data.

4.3.2 Parallel Data Management Patterns

Pack Used to eliminate unused space in a collection.

Pipeline Connects tasks in a producer-consumer manner.

Geometric Decomposition Arranges data into subcollections. Overlapping and non-overlapping compositions are possible.

Gather Reads a collection of data given a collection of indices.

Scatter The inverse of gather, given a collection of indices, it writes to such indices.

5 Parallel Models & Dependencies

5.1 Parallelism, Correctness & Dependencies

Parallel execution shall always be constrained by the sequence of operations needed to be performed for a correct result. Such execution must address control, data and system dependencies.

A *dependency* arises when one operation depends on an earlier operation to complete and produce a result before this later operation can be performed.

5.1.1 Sequential Consistency

Sequential consistency implies that the execution of two statements do not interfere with each other, meaning their running order is irrelevant. This means that the statements are *independent*, if the order of execution affects the computation outcome they are deemed *dependent*.

5.1.2 Dependencies

True Dependency A statement $S2$ has a true dependency on statement $S1$ if and only if $S2$ reads a value written by $S1$. This dependency is also known as Read After Write (RAW).

Formally we can describe a true dependency as follows:

$$out(S_1) \cap in(S_2) \neq \emptyset$$

Anti-Dependency A statement $S2$ has an anti-dependency in statement $S1$ if and only if $S2$ writes a value read by $S1$. This dependency is also known as Write After Read (WAR).

Formally we can describe an anti-dependency as follows:

$$in(S_1) \cap out(S_2) \neq \emptyset$$

Output Dependency A statement $S2$ has an output dependency on $S1$ if and only if $S2$ writes a variable written by $S1$. This dependency is also known as Write After Write (WAW).

Formally we can describe an output dependency as follows:

$$out(S_1) \cap out(S_2) \neq \emptyset$$

Loop-Carried Dependencies A loop-carried dependency is a dependency between two statement instances in two different loop iterations.

6 Synchronization

6.1 Algorithms, Programs & Processes

Sequential Algorithm A sequential algorithm is a forma description of the behavior of a sequential state machine.

Concurrent Algorithm The description of a set of sequential states machines that cooperate through a communication medium is called a concurrent algorithm.

Program When the algorithm is written in a specific programming language.

Process The running instance of an algorithm and thus of a program as well.

6.2 Process Synchronization

Process synchronization occurs when the progress of one or several processes depends on the behavior of other processes. Two types of process interaction require synchronization, competition and cooperation.

6.2.1 Competition

Competition occurs when processes compete to execute some statements and only one process at a time is allowed to execute them.

6.2.2 Cooperation

Cooperation occurs when one process can only progress after some event on another process.

Barrier A barrier is a set of control points, one per process involved in the barrier, such that each process is allowed to pass its control point only when all other processes have attained their own control points. From an operation point of view, each process has to stop until all other processes have arrived at their control point.

6.2.3 The Producer/Consumer Problem

In the producer/consumer problem we have a producer that loops forever, producing data items, and a consumer that loops forever, consuming said data items.

It is required to ensure that, only produced data items are consumed and each produced item is consumed exactly once.

Synchronization Barrier One approach to the problem consists in using a synchronization barrier to ensure that only when a data item is produced the consumer is able to consume it. While this approach works it is far from efficient.

Shared Buffer The buffer has size greater than 1 and the underlying structure can be a queue or a circular array. Whenever the producer has a new item, it adds the item at the end of the structure, the consumer then withdraws items from the head of the structure.

This way the producer only waits whenever the structure is full, the consumer will only wait whenever the structure is empty.

6.2.4 The Mutual Exclusion Problem

Critical Section A part (or several parts) that are required to be executed by a single process at a time.

Solution We are required to provide an entry algorithm as well as an exit algorithm, these are used to respectively enter and exit the critical section, ensuring that the critical section is only executed once.

Definition The mutual exclusion problem consists in implementing the operations to acquire and release a mutex in such a way that the following properties are always satisfied:

- **Mutual Exclusion** - at most one process at a time executes the critical section code.
- **Starvation-Freedom** - each invocation of `acquire_mutex` for any process p, made by p eventually terminates.

6.3 Solving Mutual Exclusion - Software Level

This discusses how to solve mutual exclusion from a software point of view.

6.3.1 Atomic Read/Write Registers

An atomic shared register satisfies the following properties:

- Each invocation *op* of a `read` or `write` operation:
  - Appears as if it was executed at a single point $T(op)$ of the timeline.
  - $T(op)$ is such that $T_{start}(op) \leq T(op) \leq T_{end}(op)$
  - For any two operation invocations op_1 and op_2 we have that $op_1 \neq op_1 \Rightarrow T(op_1) \neq T(op_1)$.
- Each read invocation returns the value written by the closest preceding write invocation in the sequence defined by the $T(\dots)$ instants associated with the operation invocations.

6.3.2 After You

We assume there are only two processes trying to access the critical region.

```
fn acquire_mutex(i) {
  AFTER_YOU = i;
  wait(AFTER_YOU != i);
}
```

Listing 1: Acquire Mutex Operation.

```
fn release_mutex(i) { }
```

Listing 2: Release Mutex Operation.

While this approach provides mutual exclusion, it does not guarantee progress. Furthermore this approach can cause starvation, for example, if process p_0 tries to access the critical region and the other process p_1 is busy, p_0 will not be able to access the critical region since the variable `AFTER_YOU` will not change.

6.3.3 Flag Array

```
fn acquire_mutex(i) {
  FLAG[i] = up;
  wait(FLAG[j] == down);
}
```

Listing 3: Acquire Mutex Operation.

```
fn release_mutex(i) {
  FLAG[i] = down;
}
```

Listing 4: Release Mutex Operation.

This approach provides mutual exclusion, however, once again it does not guarantee progress as it may cause a deadlock[1]. As we can see in Equation 18, if both processes are able to put their flag to `up` they will be waiting for each other indefinitely.

$$\begin{aligned} p_0 &\rightarrow \texttt{FLAG[0]} = \texttt{up} \\ p_1 &\rightarrow \texttt{FLAG[1]} = \texttt{up} \\ p_0 &\rightarrow \texttt{wait} \\ p_1 &\rightarrow \texttt{wait} \end{aligned} \tag{18}$$

6.3.4 Flag Array with Delayed Wait

```
fn acquire_mutex(i) {
  FLAG[i] = up;
  while (FLAG[j] == up) {
    FLAG[i] = down:
    sleep(random());
    FLAG[i] = up;
  }
}
```

Listing 5: Acquire Mutex Operation.

```
fn release_mutex(i) {
  FLAG[i] = down;
}
```

Listing 6: Release Mutex Operation.

This solution provides mutual exclusion but is not able to guarantee progress as well. However only in specific cases where the processes are synchronized and the result from the `random()` call is consistently the same for both processes the program will be stuck, creating a livelock[2].

This solution may also suffer from starvation, in the case that a process p_0 acquires the mutex and p_1 is waiting, it is possible for p_0 to release and re-acquire the mutex without p_1 getting access.

6.3.5 Mixed Approach

Just like the previous schemes, this one is also applicable for two processes only.

```
fn acquire_mutex(i) {
  FLAG[i] = up;
  AFTER_YOU = i;
  wait(
    FLAG[j] == down ||
    AFTER_YOU != i
  );
}
```

Listing 7: Acquire Mutex Operation.

[1] A *deadlock* happens when two processes are waiting on each other.

[2] A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing.

```
fn release_mutex(i) {
  FLAG[i] = down;
}
```

Listing 8: Release Mutex Operation.

By mixing the previous approaches we are able to achieve progress, since even if both flags are `up` the `AFTER_YOU` variable will make one proceed to the critical region. While this solution provides mutual exclusion and progress, it will only work for two processes.

6.3.6 Mixed Approach for N processes

```
fn acquire_mutex(i) {
  for idx in 1 to (n-1) {
    FLAG_LEVEL[i] = idx;
    AFTER_YOU[idx] = i;
    wait(
      forall k != i :
      FLAG_LEVEL[k] < idx ||
      AFTER_YOU[idx] != i
    );
  }
}
```

Listing 9: Acquire Mutex Operation.

```
fn release_mutex(i) {
  FLAG_LEVEL[i] = 0;
}
```

Listing 10: Release Mutex Operation.

This approach generalizes the one described in subsubsection 6.3.5, allowing for N processes to use the mutex. The intuition behind it so that whenever n processes try to access the critical region, they will go through $n - 1$ levels, the process reaching the last one will gain access to the region.

p_i is allowed to progress to level $idx + 1$ if, from its point of view, one of the following is true:

- All other processes are at a lower level ($\forall k \neq i$: `FLAG_LEVEL`[`k`] $<$ `idx`).
- It was not the last one to enter the level idx (`AFTER_YOU`[`idx`] $\neq i$).

6.4 Solving Mutual Exclusion - Hardware Level

6.4.1 The `test&set` & `reset` primitives

Consider that both primitives are atomic and X is a shared register initialized to 1.

- `test&set` will set X to 0 and return the previous value.
- `reset` will write 1 to X

```
fn acquire_mutex(i) {
  while (r != 1) {
    r = X.test&set();
  }
}
```

Listing 11: Acquire Mutex Operation.

```
fn release_mutex(i) {
  X.reset();
}
```

Listing 12: Release Mutex Operation.

The implementation is based on the following invariant:

$$X + \sum_{i=1}^{n} r_i = 1$$

6.4.2 The `swap` primitive

Consider a shared register X, the `swap(v)` primitive will assign v to X and return the previous value of X.

We can think of the previous instructions, `test&set` and `reset` as `swap` operations with fixed values, respectively, `swap(0)` and `swap(1)`.

```
fn acquire_mutex(i) {
  r = 0;
  while (r != 1) {
    r = X.swap(r);
  }
}
```

Listing 13: Acquire Mutex Operation.

```
fn release_mutex(i) {
  X.swap(r);
}
```

Listing 14: Release Mutex Operation.

The invariant from `test&set` applies to `swap` as well. Since this approach is just a re-implementation of the previous ones with a different instruction, it offers the same guarantees.

6.4.3 The `compare&swap` primitive

Let X be a shared register and `old` and `new` be two values. The `compare&swap` primitive returns a boolean value and may be defined as follows:

```
fn compare&swap(old, new) {
  if (X == old) {
    X = new;
    return true;
  } else {
    return false;
  }
}
```

Listing 15: `compare&swap` implementation.

The code above is considered to be executed atomically.

Consider X to be an atomic register initialized to 1, implementing mutual exclusion with `compare&swap` we have.

```
fn acquire_mutex(i) {
  while (r != 1) {
    r = X.compare&swap(1, 0);
  }
}
```

Listing 16: Acquire Mutex Operation.

```
fn release_mutex(i) {
  X = 1;
}
```

Listing 17: Release Mutex Operation.

This and the previous algorithms are not starvation free.

6.4.4 Deadlock & Starvation Free Algorithm

```
fn acquire_mutex(i) {
  FLAG[i] = up;
  wait(TURN == i || FLAG[TURN] == down);
  LOCK.acquire_lock(i);
}
```

Listing 18: Acquire Mutex Operation.

```
fn release_mutex(i) {
  FLAG[i] = down;
  if (FLAG[TURN] == down) {
    TURN = (TURN % n) + 1;
  }
  LOCK.release_lock();
}
```

Listing 19: Release Mutex Operation.

While this approach provides mutual exclusion, progress and no starvation, it does not provide fairness, meaning that we may wait for a long time until we acquire the lock.

6.4.5 The `fetch&add` primitive

Let X be a shared register, the primitive `fecth&add` will atomically increment the register X and return the new value. There are a couple of variants of this primitive in which the value return is the previous or the incremented value is not 1 but rather passed as an argument.

```
fn acquire_mutex(i) {
  my_turn = TICKET.fetch&add();
  while(NEXT != my_turn) { }
}
```

Listing 20: Acquire Mutex Operation.

```
fn release_mutex(i) {
  NEXT = NEXT + 1;
}
```

Listing 21: Release Mutex Operation.

While the `release_mutex` is not atomic, there is no problem since only one process will execute `release_mutex`.

6.5 Locking Strategies

6.5.1 Coarse-Grained Synchronization

Simple approach, use a single lock, methods are always executed in mutual exclusion, eliminating all concurrency within the object.

This approach wastes resources, for example, if there exists two objects and three threads, whenever an object is being used the other threads cannot use the other.

This approach causes false conflicts, only works under fault-free scenarios and ”reverts” the system into a sequential system.

6.5.2 Fine-Grained Synchronization

Split the object unto multiple independently-synchronized components.

6.5.3 Optimistic Synchronization

Check if the operation can be done, if the operation can be done, lock and recheck if the operation can still be done.

While this strategy is cheaper than hand-over-hand locking it is not starvation free and mistakes are expensive as safety may be compromised.

6.5.4 Lazy Synchronization

Make common operations fast and delay the hard work. Separate the complex operations into two phases, logical, marking the component, and physical, actually executing the operation.

6.6 Lock-Free Synchronization

6.6.1 Progress Conditions

Obstruction Freedom At any point a single thread executed in isolation will complete its operation in a bounded number of steps. All lock-free algorithms are obstruction-free.

This approach is the simplest one however it is too weak to guarantee progress.

Lock Freedom When the program threads are run sufficiently long, at least one of the threads makes progress (for a sensible definition of progress).

In most cases it is strong enough, while more complex than an obstruction free approach it is simpler than wait free approaches. With limited contention it will behave as wait free.

Wait Freedom Every operation has a bounded number of steps the algorithm will take before the operation completes. Being the stronger approach it is also the most complex.

7 Concurrency Errors

7.1 Data Races

Code is supposed to be executed atomically since there are multiple dependent instructions manipulating the same data. This creates the possibility to several possible interleavings.

```
void Bank::Deposit(int amount)
{
  int balance = getBalance();
  setBalance(balance + amount);
}
```

Listing 22: Deposit Operation.

```
void Bank::Withdraw()
{
  int balance = getBalance();
  setBalance(balance - amount);
}
```

Listing 23: Withdraw Operation.

As we can see in the above example, if the code is run in different threads the withdraw may not be reflected on the final balance, this happens because a thread (Thread 1) may read the balance, then the other thread (Thread 2) will update it, when Thread 1 continues the balance value it has is now outdated.

7.1.1 Data Race Detection

Data race detection may be executed by static or dynamic analysis. Static analysis depends on compiler features and may report false positives. Dynamic analysis happens at runtime, all reported data races, are data races.

Happens-Before Algorithm The Happens-Before algorithm defines a partial order for events in a set of concurrent threads. In a single thread, happens-before reflects the temporary order of event occurrence. Between threads, A happens before B if A is an unlock access in one threads and B is a lock access in a different thread[3].

$$if\ (a = \texttt{unlock(mu)} \land b = \texttt{lock(mu)}) \Rightarrow a \rightarrow b \tag{19}$$

Lock-Set Algorithm The algorithm may fail and the program still be data race free (false positives), this happens since the algorithm check a sufficient condition for data-race freedom. This algorithm does not check for data races but rather for a consistent lock discipline. The algorithm defines two data structures, `LocksHeld(t)`, the set of locks currently held by the thread t, which is initialized as $\texttt{LocksHeld(t)} = \emptyset$, and `LockSet(x)`, the set of locks that could potentially be protecting variable x, initialized as $\texttt{LockSet(t)} = \mathbb{U}$.

- When thread t acquires lock l we have:

$$\texttt{LocksHeld(t)} = \texttt{LocksHeld(t)} \cup \{l\}$$

- When thread t releases lock l we have:

$$\texttt{LocksHeld(t)} = \texttt{LocksHeld(t)} \setminus \{l\}$$

- When thread t accesses location x we have:

$$\texttt{LockSet(x)} = \texttt{LockSet(x)} \cap \texttt{LocksHeld(t)}$$

A data race is detected if `LockSet(x)` becomes empty.

7.2 High-Level Data Races

Also known as high-level atomicity violations. This happens when seemingly atomic code has interleavings that lead to a data race.

```
class Pair {
  synchronized int getX() {
    return pair.x:
  }

  synchronized int getY() {
    return pair.y:
  }

  synchronized void setPair(
    int x,
    int y
  ) {
    this.x = x;
    this.y = y;
  }

  boolean areEqual() {
    int x = this.getX(); // synchronized
    int y = this.getY(); // synchronized
    return x == y;
  }
}
```

Listing 24: Class `Pair` definition, notice the synchronized methods.

Seemingly the methods are well synchronized, however if a thread calls `areEqual`, it is possible for the pair class to be changed between `get` calls, running an interleaving like the following.

```
getX();
setPair(x', y');
getY(); // y is now y'
```

Listing 25: Possible incorrect interleaving. Assuming `y != y'`.

7.2.1 Stale-Value Errors

Stale-Value errors are caused by the privatization of data.

```
void setYToXTimesTwo() {
  int x = getX();
  setY(2 * x); // x may have changed
}

@Atomic
synchronized int getX() {
  return x;
}

@Atomic
synchronized int setY(int y) {
  this.y = y;
}
```

Listing 26: Stale-Value error example.

7.2.2 Detecting High-Level Data Races

A view of an atomic block B, named $V(B)$, is the set of variables accessed inside the atomic code block B. We can

[3] Data races between threads are possible if accesses to shared variables are not ordered.

define a *read view* of B as $V_R(B)$ as the set of variables read inside the atomic code block B. Analogously the *write view* of B, $V_W(B)$ is the set of variables written inside the atomic code block B.

```
public int getX() {
  return this.x;
}
```

Listing 27: The `getX` has the read view $V_R(\texttt{getX}) = \{\texttt{x}\}$ and the write view $V_W(\texttt{getX}) = \{\}$.

Direct Correlation There is a direct correlation between a read variable x and a written variable y if in the dependency graph D there is a path from x to y.

Common Correlation There is a common correlation between read variables x and y if, in a dependency graph D, there is a written variable z such that:

$$z \neq x, z \neq y : \exists(x \to z, y \to z)$$

High-Level Data Race Detection To detect high-level data races we first define *thread views*. A view is the set of variables guarded by a given lock. Let F be the set of fields of a program run, view $v \in \mathbb{P}(F)$ is a subset of F. The set of generated views $V(t) \subseteq \mathbb{P}(F)$ of a thread t is the set of all views v generated by t.

A view v_m generated by a thread is a maximal view, if and only if it is maximal with respect to set inclusion in $V(t)$:

$$\forall v \in V(t)[v_m \subseteq v \to v_m = v]$$

Let $M(t)$ be the set of all maximal views of thread t. Only two views which have fields in common can be responsible for a conflict.

Given a set of views $V(t)$ generated by t and a view v' generated by another thread, the overlapping views of t with v' are all non-empty intersections of views in $V(t)$ with v':

$$\textit{overlap}(t, v') \equiv \{v' \cap v | (v \in V(t)) \wedge (v \cap v' \neq \emptyset)\}$$

A set of views $V(t)$ is compatible with the maximal view v_m of another thread if and only if all overlapping views of t with v_m form a chain:

$$\textit{compatible}(t, v_m) \textit{ iff } \forall v_1, v_2 \in \textit{overlap}(t, v_m)[v_1 \subseteq v_2 \vee v_2 \subseteq v_1]$$

View consistency is defined as mutual compatibility between all threads: A thread is only allowed to use views that are compatible with the maximal views of all other threads.

$$\forall t_1 \neq t_2, v_m \in M(t)[\textit{compatible}(t_2, v_m)]$$

Example of an high-level data race detection:

$$\begin{aligned}
V_{max}(T_1) &= \{x, y\} \\
V_{max}(T_2) &= \{x\}, \{y\} \\
V_A &= V_{max}(T_1) \cap \{x\} \\
V_B &= V_{max}(T_1) \cap \{y\} \\
V_A \not\subseteq V_B &\wedge V_B \not\subseteq V_A
\end{aligned} \tag{20}$$

7.3 Ordering Violation

Missing or incorrect synchronization between two processes.

```
work = null;
LaunchThread(new Thread(2));
work = new Work();
```

Listing 28: Simple Producer, running on Thread 1

```
ConsumeWork(work);
```

Listing 29: Simple Consumer, running on Thread 2

It is possible for work to be consumed before it was initialized, thus the correct operation ordering for Thread 1 would be the following.

```
work = new Work();
LaunchThread(new Thread(2));
```

Listing 30: Simple Correct Producer, running on Thread 1

7.4 Unintended Sharing

This happens when state is shared between processes unintentionally.

```
void inc() {
  static int local = 0;
  local += 1;
}
```

Listing 31: Example of unintended sharing.

Since the `static` keyword is used the variable `local` will be kept among runs of the `inc` function, it also has the effect that the variable will be shared by threads running the same function.

7.5 Deadlocks

Deadlocks happen when processes wait on each other, this can be expressed as a cycle in a task dependency graph.

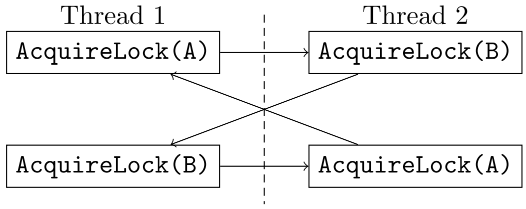

Figure 7: Deadlock dependency graph.

7.5.1 System Model

- We have a finite number of resources.
- Resources are organized into classes.
- Processes compete for accessing resources.
- If a process requests an instance of a resource class, any instance of that class must satisfy the process.

7.5.2 Resource Usage Protocol

- **Request** - the process either gets an instance of the resource immediately or waits until one is available.
- **Use** - the process can operate on its resource instance.
- **Release** - the process releases its resource instance.

7.5.3 Deadlock Definition

Informal Definition A set of two or more processes are deadlocked if:

- They are blocked.
- Each is holding a resource.
- Each is waiting to acquire a resource held by another process in the set.

Formal Definition In a formal way the conditions necessary for a deadlock are the following:

- **Mutual Exclusion** - only one process can use a resource at a time.
- **Hold & Wait** - a process holding at least one resource is waiting to acquire additional resources which are currently held by other processes.
- **No Preemption** - a resource can only be released voluntarily by the process holding it.
- **Circular Wait** - a cycle of process requests exists ($P_0 \rightarrow P_1 \rightarrow \dots P_{n-1} \rightarrow P_0$).

7.5.4 Deadlock Prevention

Restrict the way requests can be made.

Mutual Exclusion Not required for shared resources, however it must hold for non-shareable files.

Hold & Wait Must guarantee that whenever a process requests a resource, it does not hold any other resources. Require the process to request and allocate all its resources before it begins execution. Low resource utilization; possibly resulting in starvation.

No Preemption If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released. Preempted resources are added to the list of resources for which the process is waiting. Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Circular Wait Impose a total ordering of all resource types, requiring that each process requests resources in an increasing order of enumeration.

7.5.5 Deadlock Avoidance

Requires the system to have some additional *à priori* information available.

- Requires that each process declare the maximum number of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation state is defined by the number of available and allocated resources and the maximum demands of the processes.

Banker's Algorithm In the banker's algorithm processes declare their maximum resource usage of each type of resource. This number should not exceed the total number of resources in the system. Whenever a process requests resources the system must determine if the allocation of such resources will leave the system in a safe state. If it does leave the system in a safe state, the resources are allocated, otherwise the process must wait until enough resources are available.

Several data structures are required for the banker's algorithm, these data structures encode the resource-allocation state. In the following data structures, n is the number of processes in the system and m is the number of resource types.

- **Available** - A vector of length m indicates the number of available resources of each type. If **Available**$[j]$ equals k, then k instances of resource type R_i are available.
- **Max** - An $n \times m$ matrix defines the maximum demand of each process. If **Max**$[i][j]$ equals k, then process P_i may request at most k instances of resource type R_j.
- **Allocation** - An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If **Allocation**$[i][j]$ equals k, then process P_i is currently allocated k instances of resource type R_j
- **Need** - An $n \times m$ matrix indicates the remaining resource need of each process. If **Need**$[i][j]$ equals k, then process P_i may need k more instances of resource type R_j to complete its task. Note that **Need**$[i][j]$ = **Max**$[i][j]$ − **Allocation**$[i][j]$.

The *safety check* algorithm works as follows:

1. Let **Work** and **Finish** be vectors of length m and n, respectively. Initialize **Work** = **Available** and **Finish** = $false$ for $i = 0, 1, \dots, n-1$.
2. Find an index i such that both:
   (a) **Finish**$[i]$ == $false$
   (b) **Need**$_i$ $\leq$ **Work**

   If no such i exists, go to step 4.
3. **Work** = **Work** + **Allocation**$_i$
   Finish$[i]$ = $true$
   Go to step 2.
4. If **Finish**$[i]$ == $true$ for all i, then the system is in a safe state.

For the *resource requesting* algorithm, let **Request**$_i$ be the request vector for process P_i. If **Request**$_i[j]$ == k, then process P_i wants k instances of resource type R_j. The algorithm then works as follows:

1. If **Request**$_i$ $\leq$ **Need**$_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If **Request**$_i$ $\leq$ **Available**, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by, modifying the state as follows:
   Available = **Available** − **Request**$_i$

$\mathbf{Allocation}_i = \mathbf{Allocation}_i + \mathbf{Request}_i$
$\mathbf{Need}_i = \mathbf{Need}_i - \mathbf{Request}_i$
If the resulting resource-allocation state if safe, the transaction is complete, and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for $\mathbf{Request}_i$, and the old resource-allocation state is restored.

8 Labs

8.1 C Threads

To compile programs with threads in C, we must use pthread library. Lets see an example of code:

```
...

void *myThreadFun(void *vargp)
{
    sleep(1);
    printf("Hello World\n");
    return NULL;
}

int main()
{
    pthread_t thread_id;
    pthread_create(&thread_id, NULL,
     myThreadFun, NULL);
    pthread_join(thread_id, NULL);
    exit(0);
}
```

Listing 32: C Threads.

We must first declare a pthread_t variable to save a struct with some thread internals.

Next we can execute a function in a new thrad by calling *pthread_create*. If we want to wait for a thread or join the result we may use *pthread_join*.

pthread_create() takes 4 arguments:

- The first argument is a pointer to thread_id which is set by this function. (pthread_t)
- The second argument specifies attributes. If the value is NULL, then default attributes shall be used.
- The third argument is name of function (pointer) to be executed for the thread to be created.
- The fourth argument is used to pass arguments to the function.

pthread_join() takes 2 arguments:

- The first argument is a pointer to thread_id.
- The second argument is a pointer to the return values.

8.2 Monte Carlo Method

Monte Carlo methods, are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results.

In the case of our labs we have used the Monte Carlo method to approximate the π constant. We did this with threads so that we could get more samples in less time. When all the threads finished sampling, they merged the results into the main thread. This calculation is nearly embarrasing parallel.

We have implemented in Java , C and c with OpenMP

8.3 Conway's Game of Life

The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves.

In order to improve the performance of this game, we parallelized using OpenMP. Because each task is embarrasing parallel (read from a oldboard and write on a newboard), we only needed to use *#pragma omp parallel for*.

It was asked us to to other improvements such as:

- Chnage this program to use Terminal ANSI Escape code
- Include new optional flag "-p" to make a pause for the given number of miliseconds
- Include new flag "-q" to print only the last board
- Experiment different Board Sizes (speedups, efficiency, cost)

8.4 Parallel Sort

We have implemented a Parallel Quick Sort spawning tasks with *#pragma omp task*. The omp task must be inside a *#pragma omp parallel* and a *#pragma omp single* so that only one thread executes the creation of tasks but every thread can execute the tasks.

8.5 To be continued...