CP COURSE NOTES

Parallel Architectures

Flynn's Taxonomy

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

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

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

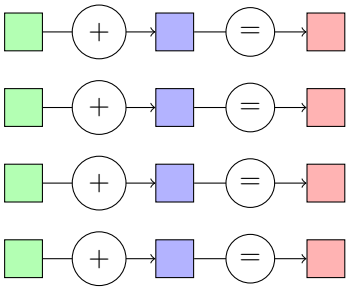

Figure 2: SISD Processing Example. One instruction is applied to each data item.

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.

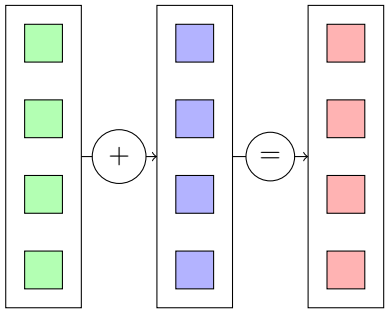

Figure 3: SIMD Processing Example. One instruction is applied to a block of data items.

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.

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.

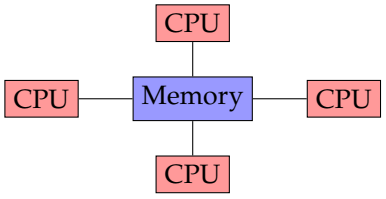

Figure 4: Uniform Memory Access (UMA). This architecture suffers from the need of synchronization among memory accesses. We can picture this as several CPUs accessing common RAM.

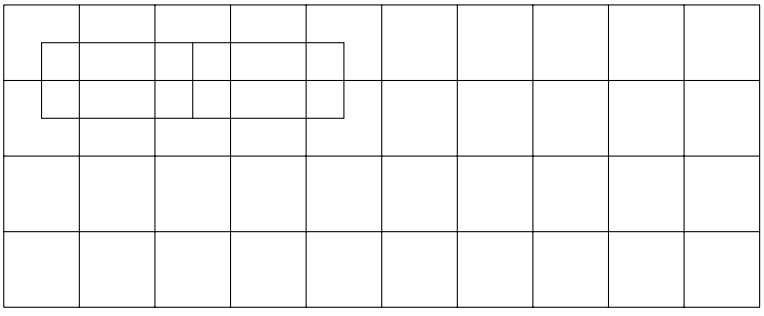

Figure 5: Non-Uniform Memory Access (NUMA). Common in distributed architectures.

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.

Memory

Memory can be split into shared memory or distributed memory.

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.

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.

Parallel Performance

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.

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

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.

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.

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.

Evaluation To evaluate scalability we can start from the following metrics.

- Sequential Runtime (T_{seq}) which is a function of problem size and architecture.
- Parallel Runtime (T_{par}) which is a function of problem size and parallel architecture, that is the number of processors used in execution.

With that in mind we can define the speedup S_p as Equation 2, efficiency E_p as Equation 3 and finally the cost C_p as Equation 4, where p is the number of available processors and T_p is the execution time on a p processor system[1].

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

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

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

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

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:

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

Gustafson-Barsis' Law

...SPEEDUP SHOULD BE MEASURED BY SCALING THE PROBLEM TO THE NUMBER OF PROCESSORS, NOT BY FIXING THE PROBLEM SIZE. - JOHN GUSTAFSON

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?

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

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.

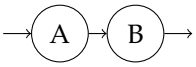

Figure 6: Serial Composition

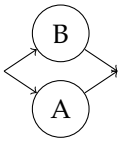

Figure 7: Parallel Composition

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.

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

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).

Parallelism

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

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

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

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

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.

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.

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.

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.

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.

Implementation Environment

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.

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.

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.

Control Patterns

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.

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.

Data Management Patterns

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.

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.

Parallel Models & Dependencies

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.

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*.

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.

Synchronization

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.

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.

Competition

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

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.

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.

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** - for any process p each invocation of `acquire_mutex` by p eventually terminates.

Bibliography