CPA NOTES

UNIT 1: INTRODUCTION TO PARALLEL COMPUTING

1.-INTRODUCTION

1.1.-PARALLEL COMPUTING APPLICATIONS

The computation performed on a **parallel computer** can execute **concurrently** several instructions for the resolution of a **single problem**, as opposed to a **sequential computer**, that instructions are executed sequentially one after the other.

Parallel computing requires:

- Redesigning the **algorithms**.
- Changing the **data structures**.

Both things are **parallel programming**.

Sequential computing does not cover the needs of:

- **Large scale** problems.
- **Real time** constraints.

Examples of the necessity of parallel computing are:

- **Simulation**: the emulation of a physical system using a computer.
- **Data processing**.

1.2.-SUPERCOMPUTING

Parallel computing, High Performance Computing or **Supercomputing**, consist in the concurrent execution of different tasks by different processing units.

Concurrency may happen at **different scales**:

- In a **single processor**:
  - Include several compute units in a single processor.
  - Simultaneous execution of multiple basic instructions.
- In a **single computer**: integration of different processors in a single computer. Depending on the coupling of processors:
  - Shared memory.
  - Distributed memory.
- On the **Internet**.

In **shared memory**, all processors can access the whole memory. Its features are:

- **Different latencies** depending on the memory bank accessed (due to proximity).
- **Scalability** up to hundreds of processors.
- **High cost** and **high performance** in fine-grain parallelism.

In distributed memory, each processor has only access to its local memory block. Its features are:

- **Information** is explicitly **exchanged** through **messages**.
- **Higher scalability** (thousands of processors).
- **Reduced cost** and **worse performance** in the fine grain.

2.-PARALLEL COMPUTING MODELS

2.1.-PARALLEL COMPUTING ARCHITECTURES

Flynn taxonomy:

- **SISD**: Single Instruction Single Data.
- **SIMD**: Single Instruction Multiple Data.
- MISD: Multiple Instruction Single Data.
- **MIMD**: Multiple Instruction Multiple Data:

A **shared memory architecture** has a single address space for all the processors. Types:

- UMA: Uniform Memory Access.
- NUMA: Non-Uniform Memory Access.
- cc-NUMA: Cache Coherent NUMA.

The main **advantage** is that it is **easy to program**, but the **disadvantages** are that is has **low scalability** and **high price**.

A **distributed memory architecture** requires a communication network to let processors access data outside the local space. Its features are:

- There is no global memory concept.
- Independent processors (no coherence).
- The programmer explicitly exchanges data.

The **advantages** are that is has **high scalability** and **low cost**, but the main **disadvantage** is that is **difficult to program**.

A **hybrid architecture** (distributed-shared memory) is a combination of both models seen. Its features are:

- Each node is a multiprocessor.
- Communication to move data from one node to another.

Multi-core processors are the current trend in processor design. Its features are:

- Symmetric (or not) multiprocessing on a single chip.
- Several cache levels in the same chip.

The main **advantage** is the **low cost**, but the main **disadvantage** is that it has **low efficiency**.

Many-cores processors are massively parallel and with a large number of simple cores. Its features are:

- Many cores (in the order of thousands).
- Light threads, very quick context switch.

The **advantages** are the **low cost** and **high performance**, but the main **disadvantage** is that is **difficult to program**.

A **cluster** is simply a set of PCs or workstations connected in a network to run parallel programs. Currently, a commercial cluster typically comprises:

- Rack structure.
- A set of nodes.
- Network infrastructure.
- A front-end node.

2.2.-PARALLEL PROGRAMMING METHODOLOGIES

Different methodologies can be applied:

- Automatic parallelization (parallelizing compiler).
- Semi-automatic parallelization (compiler directives).
- New programming languages.
- Extensions of conventional languages.
- Libraries of subroutines or functions (API).

UNIT 2: SHARED MEMORY. BASIC PARALLEL ALGORITHMS DESIGN

1.-SHARED MEMORY MODEL

Concurrent processes are typically defined using **fork-join**-like constructions. Fork creates a new concurrent task that starts its execution at the same point where the parent task made the fork. Join waits for the task to finish.

This scheme can be implemented at the level of:

- Operating system processes (heavy processes).
- Threads (light processes).

The **shared memory model** has the following features:

- Tasks share a common memory-address space.
- Programming quite similar to sequential case.
    - Any data are accessible by all.
    - No need to exchange data explicitly.

Some **drawbacks** are:

- Concurrent memory access may be problematic.
- Need to be coordinated.
- Unpredictable results if data access is not properly protected.
- Data locality is difficult to control (cache memories).

A **thread** is an independent instruction flow that can be scheduled for execution by the operating system.

The thread model is closely related to the shared memory model and has the following **characteristics**:

- A process may have **multiple concurrent execution threads**.
- **Each thread** has "**private**" **data**.
- Threads **share resources/memory of the process**.
- **Synchronization** is needed.

Examples of threads:

- Java thread.
- POSIX thread (pthread).
- OpenMP

In OpenMP, the creation and termination of threads is implicit in some directives, in other words, the programmer does not bother about explicit fork/join.

In **Unix processes**, each process contains information about resources and its execution status:

- Executable code (read-only, can be shared).
- Variables (global, heap and stack).
- Execution context.
- System resources:
    - Identifiers (process, user, group).
    - Environment, work directory, signals.
    - File descriptors.

In **multi-threaded processes**:

- Each thread has its own execution context.
- Each thread has its own independent stack.
- System resources are shared.

In the memory model of Unix processes, the information is in the kernel of the operating system (PCB: Process Control Back).

There are two models of memory model with threads:

- **Simple model**: single address space.
- **More realistic model**: single address space, with private variables for each thread.

Each thread has its own stack. Some variables are created in the stack (local variables). A thread cannot know if another thread's stack is active.

The **exchange of inform**ation among threads is performed by reading and writing on variables in the shared memory space.

Simultaneous access can produce a **race condition**:

- Final result could be incorrect.
- Nondeterministic nature.

To **solve race conditions**, we use:

- **Atomic operations**:
    - Force problematic operations to be performed atomically.
    - Special instructions of the processor.
- **Critical sections**:
    - Code fragments with more than one instruction.
    - Only one thread can execute the section simultaneously.
    - It requires synchronization mechanisms.
    - Risk of deadlocks.

2.-FUNDAMENTALS OF PARALLEL ALGORITHM DESIGN

2.1.-DEPENDENCY ANALYSIS

Parallelizing an algorithm implies finding concurrent tasks (parts of the algorithm that can be run in parallel).

It is possible to determine if there exist dependencies between two tasks from the input/output data of each task.

Bernstein conditions: two tasks T_i and T_j (T_i precedes T_j sequentially) are independent if:

1. $I_j \cap O_i = \emptyset$
2. $I_i \cap O_j = \emptyset$
3. $O_i \cap O_j = \emptyset$

I_i and O_i stand for the set of variables read and written by T_i.

Dependency types:

- **Flow dependencies** (condition 1 is not fulfilled).
- **Anti-dependency** (condition 2 is not fulfilled).
- **Output dependency** (condition 3 is not fulfilled).

Sometimes data dependencies may be eliminated modifying the algorithm.

General idea for the design of parallel algorithms:

1. **Task decomposition**. Requires a detailed analysis of the problem (Task Dependency Graph).
2. **Task assignment**. Which thread/process executes each task. Often implies agglomeration of several tasks.

Usually there are several possible parallelization strategies. Using one decomposition or another may have a great impact on performance. We must try to maximize the degree of concurrency.

2.2.-DEPENDENCY GRAPH

The **Task Dependency Graph** is an abstraction used to express the dependencies among the tasks and their relative execution order:

- It is a Directed Acyclic Graph (DAG).
- Nodes denote the tasks (may have an associated cost).
- Edges represent the dependencies among tasks.

Definitions:

- **Length of a path**: sum of the costs ci of each node contained in the path
- **Critical path**: longest path between a starting and a final node.
- **Maximum concurrency degree**: larger number of tasks that can be executed concurrently.
- **Average concurrency degree**:

$$M = \sum_{i=1}^{N} \frac{C_i}{L}$$

(N = total nodes, L = length of the critical path)

Sometimes the graph incorporates information related to **communication**:

- Possibility of adding auxiliary nodes (without cost).
- Edges with weight: denote the communication between tasks (value proportional to the amount of data).

3.-PERFORMANCE EVALUATION

The main objective of parallel computing is to **increase performance**:

- Is very important to know how the different parts of a parallel program behave.
- Is also important to know how they will behave when the number of processors and the size of the program change.

There are **two types of analysis**:

- A **priori analysis**:
    - It is performed on the pseudocode and the program design, before the implementation of a program.
    - Independent of the machine where it is executed.
    - Allows to identify the best approach to implement a parallel program.
    - Allows to determine the best size of the problem and the features of the hardware used.
- A **posteriori analysis**:
    - It is performed on a specific implementation and machine and using a defined set of input data.
    - Allows to analyse bottlenecks and detect conditions not foreseen during the design.

In a **theoretical analysis**, the cost is analysed in terms of the problem size (n). In many cases the cost depends only on n: t(n). But sometimes, given the same problem size n, different behaviour may be observed depending on the input data:

- Cost of the best case
- Cost of the worst case
- Average cost

In practice, asymptotic bounds are used (lower and upper).

A **Flop** is a floating point operation and it is the measurement unit for:

- Cost of algorithms.
- Performance of computers (flop/s).

The **cost of integer operations** is considered **negligible**. The cost of other operations in floating point is expressed in terms of the Flop unit. The flop represents a machine-independent cost measurement unit (the time elapsed in a flop varies from one processor to another).

Asymptotic notation:

- **Big O notation**, O:
  - Defines an (asymptotic) upper bound for the growth of a function, disregarding constants.
  - In practice, it is the highest-order term of the cost expression without considering its coefficient.
- **Small O notation**, o:
  - Also takes into account the coefficient of the highest-order term.
  - Appropriate to compare two algorithms of the same O order.

Parameters to evaluate the performance:

- **Absolute parameters**:
  - Allow us to know the real cost of parallel algorithms.
  - They are the basis for the computation of relative parameters that are used to compare algorithms.
  - They are the most important ones for real-time problems.
- **Relative parameters**:
  - Allow us to compare parallel algorithms among them and with respect to the sequential implementation.
  - They provide information about the degree of utilization of processors.

3.1.-ABSOLUTE PARAMETERS

Absolute parameters:

- **Execution time of a sequential algorithm**: t(n)
- **Execution time of a parallel algorithm**: t(n, p)
  - **Arithmetic time**: $t_a(n, p)$
  - **Communication time**: $t_c(n, p)$
- **Total Cost**: C(n, p)
- **Overhead**: $t_o(n, p)$

When the problem size is always n, without ambiguity, it will be omitted. Sometimes we will use subindices instead of functions.

The **execution time** is the time spent in the execution by the sequential algorithm (using only one processor, t(n)) or by the parallel algorithm (in p processors, t(n, p)). The a priori cost is measured in flops. We will take into account only the number of floating point operations. Experimentally the cost will be measured in seconds.

Useful expressions for computing the cost:

$$\sum_{i=1}^{n} 1 = n \qquad \sum_{i=1}^{n} i \approx \frac{n^2}{2} \qquad \sum_{i=1}^{n} i^2 \approx \frac{n^3}{3}$$

The execution of a parallel algorithm normally implies an extra time with respect to the sequential algorithm. The parallel **total cost** accounts for the total time employed by a parallel algorithm:

$$C(n, p) = p \cdot t(n, p)$$

The **overhead** indicates which is the added cost with respect to the sequential algorithm:

$$t_o(n, p) = C(n, p) - t(n)$$

The **speedup** indicates the speed gain of a parallel algorithm with respect to its sequential version:

$$S(n,p) = \frac{t(n)}{t(n,p)}$$

The **reference time t(n)** could be:

- The best sequential algorithm.
- The parallel algorithm run on 1 processor.

The **efficiency** measures the degree of utilization of the parallel computer by the parallel algorithm:

$$E(n,p) = \frac{S(n,p)}{p}$$

Usually expressed as a percentage (or parts per unit).

Ideally, for p processors we have a speedup equal to p (efficiency equal to 1). The factors that determine that we get more or less close to this ideal speedup are:

- Appropriate **parallelization des**ign.
  - Well balanced load distribution.
  - Minimize time in which processors are idle.
  - Minimum possible overhead.
- Specific aspects of the **architecture where it runs**.
  - Different in shared memory or message passing.
  - **Data access time** is not considered in the theoretical cost analysis, but it is very important in current architectures.

Synchronization may have a negative impact on efficiency. The critical section should be as small as possible. Otherwise a “serialization” occurs. In the same way, **barriers** should be used only when necessary.

4.-ALGORITHM DESIGN: TASK DECOMPOSITION

Parallel algorithms have a higher design complexity than sequential ones:

- Concurrency (implies communication and synchronization).
- Assignment of data and code to processors.
- Concurrent access to shared data.
- Scalability for an increasing number of processors.

Main steps in the design are:

- Task decomposition.
- Task assignment.

A **task** is each of the computation units defined by the programmer that can potentially be executed in parallel. The process of splitting a computation/program in tasks is called **decomposition**.

The decomposition can be **fine-grained** or **coarse-grained**. Usually a fine-grained decomposition is performed, and later tasks are grouped together into coarser tasks.

The **decomposition techniques** are:

- Domain decomposition.
- Functional decomposition driven by data flow.
- Recursive decomposition.
- Other: exploratory decomposition, speculative decomposition, mixed approaches.

4.1.-DOMAIN DECOMPOSITION

In case of large, regular data structures, **data** are **split in chunks** of similar size (sub-domains) and a **task** is **assigned** to **each sub-domain**, which will perform the required operations on the sub-domain's data.

Domain decomposition is typically used when it is possible to apply the same set of operations on the data of every sub-domain.

The **decomposition** can be:

- Centred on output data.
- Centred on input data.
- Centred on intermediate data.
- Block-oriented decompositions (matrix algorithms).

When it is **centred on output data**, each component of the output data can be computed independently from the rest.

When it is **centred on input data**, each component of the input data can be computed independently from the rest.

4.2.-OTHER DECOMPOSITIONS

The **functional decomposition** driven by data flow is used when:

- The resolution of the problem can be split into phases.
- Each phase executes a different algorithm.

Typically, it involves the next **steps**:

1. The different phases are identified.
2. A task is assigned to each phase.
3. Data requirements for each task are analysed:
    a. If the data overlapping among different tasks is minimum and the data flow among them is relatively small, the decomposition will be complete and feasible.
    b. Otherwise, a different decomposition approach may be needed.

The recursive decomposition is a method to obtain concurrency in problems that can be solved using the **divide and conquer technique**:

1. Divide the original problem in two or more subproblems.
2. In turn, these subproblems are divided in two or more subproblems, and so on until a base case is reached.
3. Resulting data are appropriately combined to obtain the final result.

It can be implemented in **different forms**:

- Replicated workers with a task pool.
- Recursive algorithm.

5.-ALGORITHMIC SCHEMES

Algorithmic schemes (or templates) are commonly used on parallelization approaches. A scheme can be used to solve a wide range of problems. A problem might require combining several schemes.

Some **schemes**:

- Data parallelism / data partitioning.
- Task parallelism (manager-workers, process farm, replicated workers).
- Tree and graph based schemes (divide and conquer).
- Segmented parallelism (pipelining).
- Synchronous parallelism.

5.1.-REPLICATED WORKERS

A **task pool** is a shared data structure containing pending tasks.

5.2.-DIVIDE AND CONQUER

Divide and Conquer consists in solving a problem by splitting it into a series of similar sub-problems, solving these sub-problems and combining theirs solutions. It is typically implemented in a recursive way (tree).

There are several **types of tasks**:

- **Dividing the problem**: it is performed in the inner nodes to create child nodes.
- **Solving the base case**: only in the leaves of the tree.
- **Combining the results**: performed in the inner nodes, collapsing the associated sub-tree.

SEMINAR 2: PROGRAMMING WITH OPENMP

1.-BASIC CONCEPTS

1.1.-PROGRAMMING MODEL

OpenMP programming is mainly based on **compiler directives**. Main advantages:

- It eases the adaptation (the compiler ignores #pragma).
- It enables incremental parallelization.
- It enables compiler-based optimization.

OpenMP execution models follows the **fork-join scheme**:

1. Main thread executes the sequential part.
2. Master thread reaches the directive, creates threads.
3. All threads execute concurrently.
4. Implicit barrier: all threads wait for completion.
5. Main thread continues, the rest remain idle.

There are directives to create threads and sharing the work. Directives define **parallel regions**. Other directives/clauses:

- **Scope of variable**: private, shared, reduction.
- **Synchronization**: critical, barrier.

In OpenMP, idle threads are not destroyed, they remain ready for the next parallel region. Threads created by a directive are called **team**. Each thread has its own execution context (including the stack).

The syntax of **directives** is: **#pragma omp <directive> [clause [...]]**.

The syntax for using functions is:

```
#include <omp.h>

...

iam = omp_get_thread_num();
```

The number of threads can be specified:

- Using the **num_threads** clause.
- Calling the **omp_set_num_threads()** function before the parallel region.
- At run time, with **OMP_NUM_THREADS**.

Useful functions:

- **omp_get_num_threads()**: returns the number of threads
- **omp_get_thread_num()**: returns the identifier of the thread (starting from 0, main thread is always 0)

2.-LOOP PARALLELIZATION

2.1.-PARALLEL FOR DIRECTIVE

The loop below the directive is parallelized. OpenMP imposes restrictions to the loop type. Variable with concurrent access, but no more than one thread accessing the same position, doesn't need any further action. On the other hand, variables with different values in each thread must be private.

2.2.-VARIABLE SCOPE

Variables are classified according to their **scope**:

- **Private**: each thread has a different replica.
- **Shared**: all threads can read and write.

A typical source of errors is choosing an incorrect scope. The scope can be modified with clauses added to the directives:

- private, shared.
- reduction.
- firstprivate, lastprivate.

If the scope of a variable is not specified, by default it is shared.

Exceptions (private):

- Index variable of the parallelized loop.
- Local variables of the called subroutines (except if they are declared static).
- Automatic variables declared inside the loop.

Clause default: default(none) forces to specify the scope of all variables.

To perform **reductions**, we have to use commutative and associative operators (+, *, -, &, |, ˆ, &&, ||, max, min) in a variable. It works as a private variable, but:

- At the end, the private values are combined.
- Correctly initialized (to the neutral element of the operation).

Private variables are created without an initial value and after the block its value is undefined:

- **firstprivate**: initializes to the value of the main thread at the begining of the block.
- **lastprivate**: the value of the variable after the block is the one of the "last" iteration of the loop.

2.3.-PERFORMANCE IMPROVEMENT

Loop parallelization introduces an overhead: activation and deactivation of threads, synchronization, etc. In simple loops, the overhead could be even larger than the compute time. In order to solve this we can use an "if clause" in the directive. If the expression is false, the loop is executed sequentially. This clause can be also used to avoid data dependencies detected at execution time.

Usually, it is recommended to **parallelize the outer loop in nested loops**. When data dependencies prevent from parallelizing the outer loop, loop exchange may be convenient.

Ideally, all iterations cost the same and each thread gets assigned approximately the same number of iterations. In reality, a workload unbalance may appear, therefore reducing performance. In OpenMP it is possible to select the scheduling. Scheduling can be:

- **Static**: iterations are assigned to threads a priori.
- **Dynamic**: assignment adapts to the current execution.

The scheduling is done on the basis of contiguous ranges of iterations (chunks).

The syntax of the scheduling clause is: **schedule(type[,chunk])**. The type can be:

- **static** (without chunk): each thread receives an iteration range of similar size.
- **static** (with chunk): cyclic (round-robin) assignment of ranges of size chunk.
- **dynamic** (optional chunk, 1 by defalut): ranges are being assigned as required (first-come, first-served).
- **guided** (optional minimum chunk): same as dynamic but the size of the iteration range decreases exponentially with the loop progress.
- **runtime**: the scheduling is defined by the value of the environment variable OMP_SCHEDULE.

3.-PARALLEL REGIONS

3.1.-PARALLEL DIRECTIVE

The block below the directive is executed in a replicated way. Some of the allowed clauses are: private, shared, default, reduction, if.

3.2.-WORK SHARING

Along with the replicated execution, it is often necessary to share the work among the threads:

- Each thread works on a part of the data structure, or.
- Each thread performs a different operation.

Possible **mechanisms for worksharing**:

- **Based on the thread identifier**.
- **Parallel task queue**.
- Using **OpenMP specific constructs**.

When we use **sharing based on thread identifier**, we us the following function in order to determine which part of the workload is done by each thread:

- **omp_get_num_threads()**: returns the number of threads.
- **omp_get_thread_num()**: returns the thread identifier.

A **parallel task queue** is a shared data structure containing a list of "tasks" to be performed. Tasks can be processed concurrently. Any task can be run by any thread.

The solutions mentioned before are quite primitive since the programmer is in charge of splitting the workload and the code in large programs is obscure and complicated.

OpenMP offers **three types of specific worksharing constructs**:

- **for construct** to split iterations of loops.
- **Sections** to distinguish different parts of the code.
- Code to be executed by a **single thread**.

There is an implicit barrier at the end of a block.

The **for construct** automatically distributes the iterations of a loop. The loop iterations are not replicated but shared among the threads. "**parallel**" and "**for**" directives can be combined in one. When several independent loops appear in the same parallel region, "**nowait**" removes the implicit barrier.

The **sections construct** is used for independent pieces of code difficult to parallelize:

- Individually they represent little work, or.
- Each fragment is inherently sequential.

A thread may execute more than one section. The clauses that can be used are: private, first/lastprivate, reduction, nowait.

The **single construct** is for code fragments that must be executed by a single thread. Some allowed clauses are: private, firstprivate, nowait.

The **master directive** identifies a code block inside a parallel region to be executed only by the master thread. The differences with "single" are:

- It does not require all threads to reach this construction.
- There is no implicit barrier (other threads simply skip this code).

4.-SYNCHRONIZATION

There are cases where concurrent access does not produce a race condition, but in general, synchronization mechanisms are needed:

- Mutual exclusion.
- Other type of synchronization.

4.1.-MUTUAL EXCLUSION

Mutual exclusion when accessing shared variables prevents any race condition.

OpenMP provides **three different constructs**:

- **Critical sections**: critical directive.
- **Atomic operations**: atomic directive.
- **Locks**: *_lock routines.

When a thread reaches the block with the **critical directive** (the critical section), it waits until no other thread is executing it at the same time. OpenMP guarantees **progress** (at least one waiting thread enters the critical section), but not **limited wait time**. By adding a name, we can have several unrelated critical sections.

Atomic operations load-modify-store. The code is much more efficient than using critical and enables updating the elements of x in parallel.

The **barrier directive** is used in order that when thread reach the barrier, they have to wait for the rest to arrive. It is used to guarantee that a phase has been finished completely before proceeding to the next phase.

The ordered directive is used to make sure that a portion of the code of the iterations is executed in the original **sequential order**. Restrictions:

- If a parallel loop includes an ordered directive, the ordered clause should also be added to the loop.
- Only one **ordered section** is allowed per iteration.