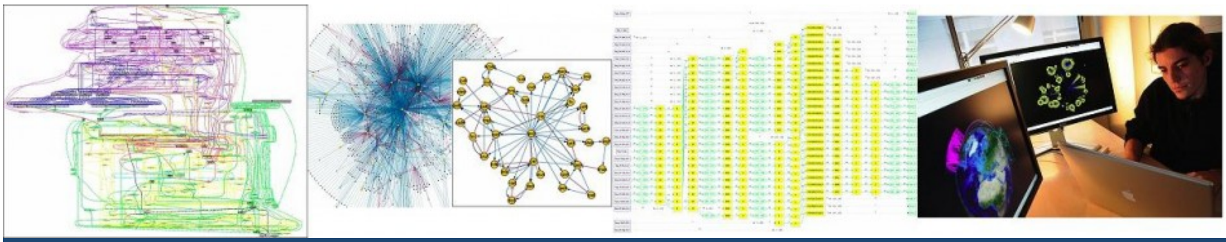


Conception of parallel algorithms

Some references

• **Parallel Programming – For Multicore and Cluster System,** T. Rauber, G. Rünger

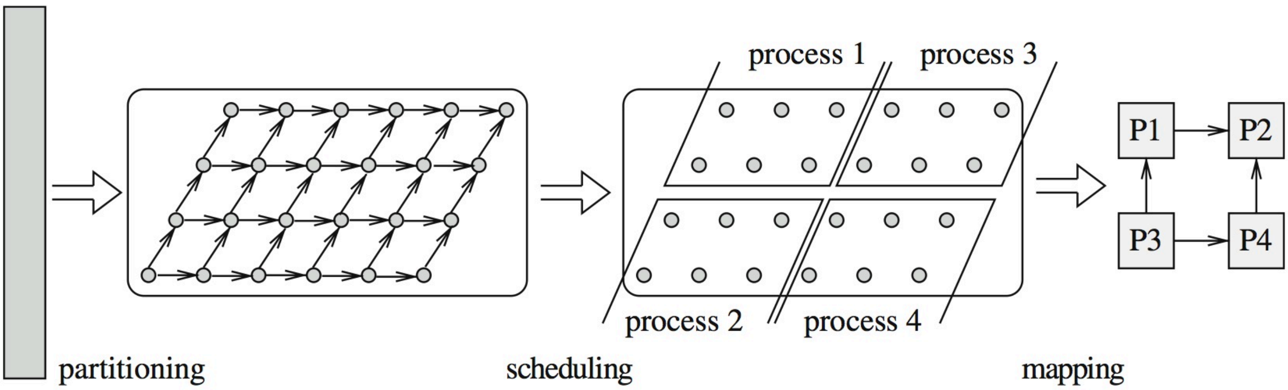
• **Decomposing the Potentially Parallel A one day course**, Elspeth Minty, Robert Davey, Alan Simpson, David Henty, Edinburgh Parallel Computing Centre The University of Edinburgh

Algorithm Parallelization

- The target architecture and the programing model have an influence over the way of designing (and writing) a parallel algorithm
- The parallel algorithm should be designed depending of the characteristics of the sequential algorithm
  - Type of data manipulated (tables, graphs, data structures, …)
  - Data accesses (regular, random, …)
  - Granularity
  - Dependences (control, data)
  - …

- **Goals**
  - To obtain the same results as the sequential program!
  - Gain some time
  - Reduce the storage consumption (memory, disk)

Parallelization Steps



Computation decomposition

- The computations of a sequential program are decomposed into tasks (with dependencies between them)
- Each task is the smallest unit of parallelism
- **Various levels of execution**
  - Instruction parallelism
  - Data parallelism
  - Functional parallelism
- The decomposition can be static (before execution) or dynamic (at runtime)
- At any time of execution, the number of "executable" tasks gives a higher bound of the degree of parallelism of the program
- The decomposition into tasks consists in generating enough parallelism to have a maximum occupancy of the cores / processors / machines

Computation decomposition, contd.

- Pay attention to **granularity** !
  - The execution time of a task must be large in relation to the time it takes to schedule it, place it, start it, acquire its data, return the computed data
  - Large coarse grain tasks with numerous computations
    - Fine-grained tasks with few computations
  - Compromise to find according to the performances (computation, access to the data, overhead of the system, communications, disk access, ...)

Assignment to processes or threads

- A **process** or **thread** represents a **flow of control** executed by a **processor** or a **core**
  - Performs tasks one after the other
  - Not necessarily the same number as the number of processors (or cores)
- Assign tasks to ensure proper balancing (i.e. all processors or cores must have the same volume of computation to run)
  - Attention should be paid to **memory access** (for shared memories) and **message exchanges** (for distributed memories)
  - Re-use of data if possible
  - Also called **scheduling**
    - If executed during the initialization phase of the program: **static scheduling**
    - If executed during program execution: **dynamic scheduling**
  - **Main goal:** equal use of processors while maintaining a smallest amount of communication between processors

Placing processes on processors

- Generally placing each process (or thread) on separate processors (or core)
- If more processes than processors then **placing** more processes on processors
  - Performed by the system and / or program directives
  - Goal: Balance and reduce communications

- **Scheduling algorithm**
  - A method for efficiently executing a set of tasks for a determined duration on a set of specified execution units
  - Satisfying dependencies between tasks (precedence constraints)
  - Capacity constraints (because number of execution units limited)
  - Sequential or Parallel Tasks
  - **Goal**
    - Set the **start time** and the target execution unit for each task
    - Minimize the **execution time** (makespan) = time between start of the 1st task and the end of the last task

Parallelism levels

- Computations executed by a program provide opportunities for parallel execution at different levels
  - Instruction
  - Loop
  - Functions

- Different levels of granularity (fine, medium, wide)
  - Possibility of grouping the different elements to increase the granularity
  - Different algorithms according to the different granularities

Parallelism at the instruction level

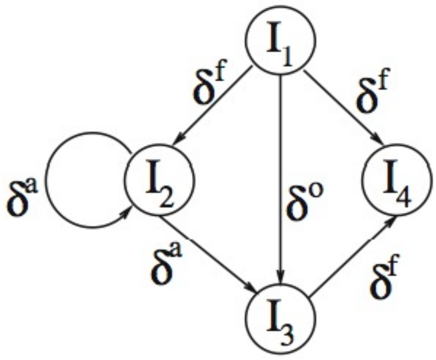
- The instructions of a program can run in parallel if they are **independent**

- There are several types of dependencies
  - **Flow dependencies** (ou true dependencies): there is a flow dependency between I_1 and I_2 if I_1 computes a result value in a register or a variable used by I_2 as an operand
  - **Anti-dependency**: there is an anti-dependency between I_1 and I_2 if I_1 uses a register or an operand variable that is used by I_2 to store a result
  - **Output dependency:** there is an output dependency between I_1 and I_2 if I_1 and I_2 use the same register or the same variable to store a result of a calculation

I_1: $\underline{R_1} \leftarrow R_2+R_3$	I_1: $R_1 \leftarrow \underline{R_2}+R_3$	I_1: $\underline{R_1} \leftarrow R_2+R_3$
I_2: $R_5 \leftarrow \underline{R_1}+R_4$	I_2: $\underline{R_2} \leftarrow R_4+R_5$	I_2: $\underline{R_1} \leftarrow R_4+R_5$
flow dependency	anti dependency	output dependency

Dependence Graph

I_1: $R_1 \leftarrow A$
I_2: $R_2 \leftarrow R_2 + R_1$
I_3: $R_1 \leftarrow R_3$
I_4: $B \leftarrow R_1$



Using instruction parallelism

- Used by superscalar processors
  - Dynamic scheduling of hardware-level instructions with extraction of independent instructions automatically
  - Then assigned to independent units
- On VLIW, static scheduling performed by the compiler with arrangement of instructions in long instructions with assignment to the units
- As an input a sequential program with no explicit parallelism

Data parallelism

- Same operation performed on different elements of a larger data structure (array for example)
  - If these operations are independent then they can be carried out in parallel
  - The elements of the data structure are distributed in a balanced manner on the processors
  - Each processor executes the operation on the elements assigned to it
- Extension of programming languages (data-parallel languages)
  - A single flow of control
  - Special constructions to express data-parallel operations on arrays (SIMD model)
  - C*, data-parallel C, PC++, DINO, High Performance Fortran, Vienna Fortran

```
a(1:n) = b(0:n-1) + c(1:n)
```

```
for (i=1:n)
   a(i) = b(i-1) + c(i)
endfor.
```

Owner computes rule

Loop parallelism

- Many algorithms traverse a large data structure (data array, matrix, image, ...)
- Expressed by a loop in the imperative languages
- If there are no dependencies between the different iterations of a loop, we will have a parallel loop
- Several kinds of parallel loops
  - FORALL
  - DOPAR
- Possibility of having
  - Multiple instructions in a loop
  - Nested loops
- Loop transformations of to find parallelism

FORALL loop

- With one or more assignments to array elements
- If only one assignment, then equivalent to an array assignment
  - The computations given in the right part of the assignment are executed in any order
  - Then the results are affected in the array elements (in any order)

```
forall (i = 1:n)
  a(i) = a(i-1) + a(i+1)
endforall
```

```
a(1:n) = a(0:n-1) + a(2:n+1)
```

- If more than one assignment
  - They are executed one after the other as array assignments
  - An assignment ends before the next one starts

DOPAR loop

- Can contain
  - one or more assignments to tables
  - other instructions or other loops
- Executed by multiple processors in parallel (in any order)
- The instructions for each iteration are executed sequentially in the order of the program, using the values of the variables in the initial state (before execution of the DOPAR loop)
- The updates of the variables of an iteration are not visible for the other iterations
- Once all iterations have been executed, the iterations updates are combined and a new global state is computed

Execution Comparison

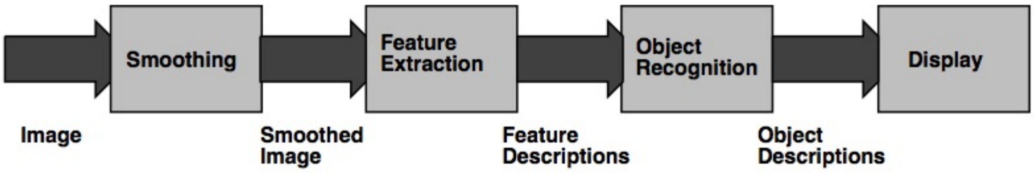
```
for (i=1:4)            forall (i=1:4)         dopar (i=1:4)
  a(i)=a(i)+1            a(i)=a(i)+1            a(i)=a(i)+1
  b(i)=a(i-1)+a(i+1)     b(i)=a(i-1)+a(i+1)     b(i)=a(i-1)+a(i+1)
endfor                 endforall              enddopar
```

Start values			After for loop	After forall loop	After dopar loop
a(0)	1				
a(1)	2	b(1)	4	5	4
a(2)	3	b(2)	7	8	6
a(3)	4	b(3)	9	10	8
a(4)	5	b(4)	11	11	10
a(5)	6				

Functional parallelism

- Many parallel programs have independent parts that can be executed in parallel
  - Simple Instructions, blocks, loops, or function calls
- One can see the parallelism of tasks by considering the different parts of a program as tasks
  - **Functional parallelism** or **task parallelism**
- To use task parallelism, tasks and their dependencies are represented as a graph where the **nodes** are **tasks** and the **vertexes dependencies between tasks**
  - **Sequential** tasks or **parallel** tasks (mixed parallelism)
- To schedule tasks, you must assign a start date to each task that satisfies dependencies (a task can not be started until all tasks on which it depends have been completed)
  - Minimize overall execution time (makespan)
  - Static and dynamic algorithms

Functional parallelism



Limitations

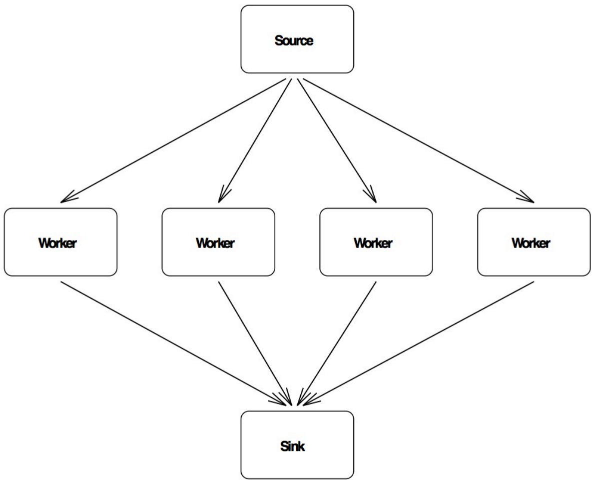
- **Start-up costs**
  - At the beginning (and at the end of the pipeline) few active tasks
- **Load Balancing**
  - If tasks have higher costs, it is difficult to balance the work between processors (think of re-splitting if possible)
- **Scalability**
  - Limitation on the number of processors that can be used

Functional parallelism

- **Static Scheduling Algorithm**
  - Determines assignment of tasks deterministically at the start of the program or at compilation
  - Based on an estimate of the execution time of tasks (measures or models)
- **Dynamic Scheduling Algorithm**
  - Determines the assignment of tasks at runtime
  - Allows to adapt placement according to conditions
  - Using a task pool that contains tasks that are ready to run
    - At each task execution, all tasks that depend on them (and whose dependencies are satisfied) are found in the pool
    - Often used in shared memory machines (pool in main memory)
- Languages that allow functional parallelism
  - TwoL, PCN, P3L, Kaapi, OpenMP
- Used in multi-threaded systems, workflows systems

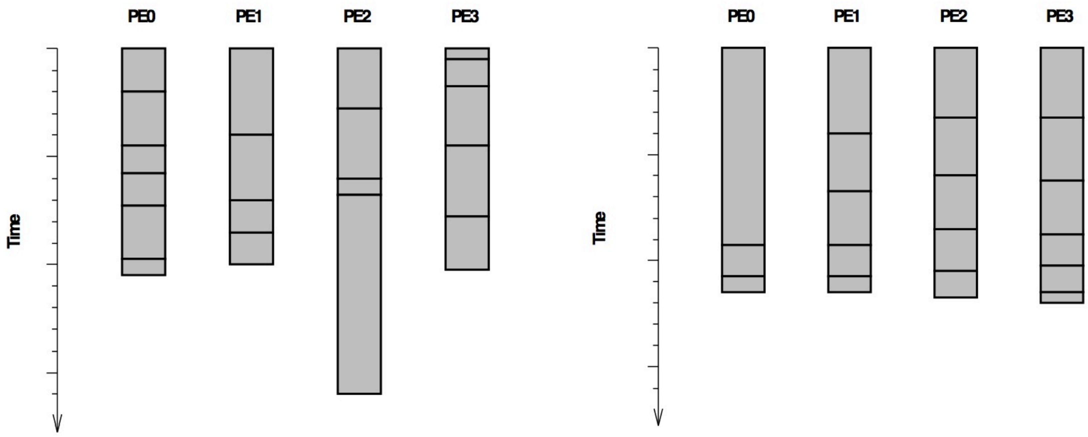
Task farming

- **Source**
  - Divides initial tasks among workers and ensures balancing
- **Worker**
  - Receives the task from the source, performs the job and passes the result
- **Sink**
  - Receives completed tasks from workers and collects partial results



Task farming, contd.

- **Limitations of the task farming model**
  - Large number of communications
  - Limited to certain types of applications
  - Scheduling management if no knowledge of task execution times



Conclusion

- Not one single solution but several solutions
- Combination of models
  - Task parallelism and data parallelism (mixed parallelism)
- Implicit or explicit parallelism
- Automatic or semi-automatic discovery of parallelism
- Scheduling and task load-balancing algorithms
- Importance of execution models!