UNIVERSITÀ di VERONA

Dipartimento di INFORMATICA

PARCO Parallel Computing Lab

OpenMP

Part I

Laurea magistrale in Ingegneria e Scienze Informatiche
Laurea Magistrale in Medical Bioinformatics

Nicola Bombieri – Federico Busato

Agenda

- Introduction to OpenMP
  - What is OpenMP?
  - OpenMP execution and memory model
- OpenMP basic syntax
  - Parallel regions
  - Loops
  - Data sharing
- Examples and exercises
- References

What is OpenMP?

*"**OpenMP** (Open Multi-Processing) is an Application Programming Interface (API) that supports multi-platform shared memory parallel programming in C/C++ and Fortran on many architectures, including Unix and Microsoft Windows platforms"*

- It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior

Creating an OpenMP program (from a sequential one)

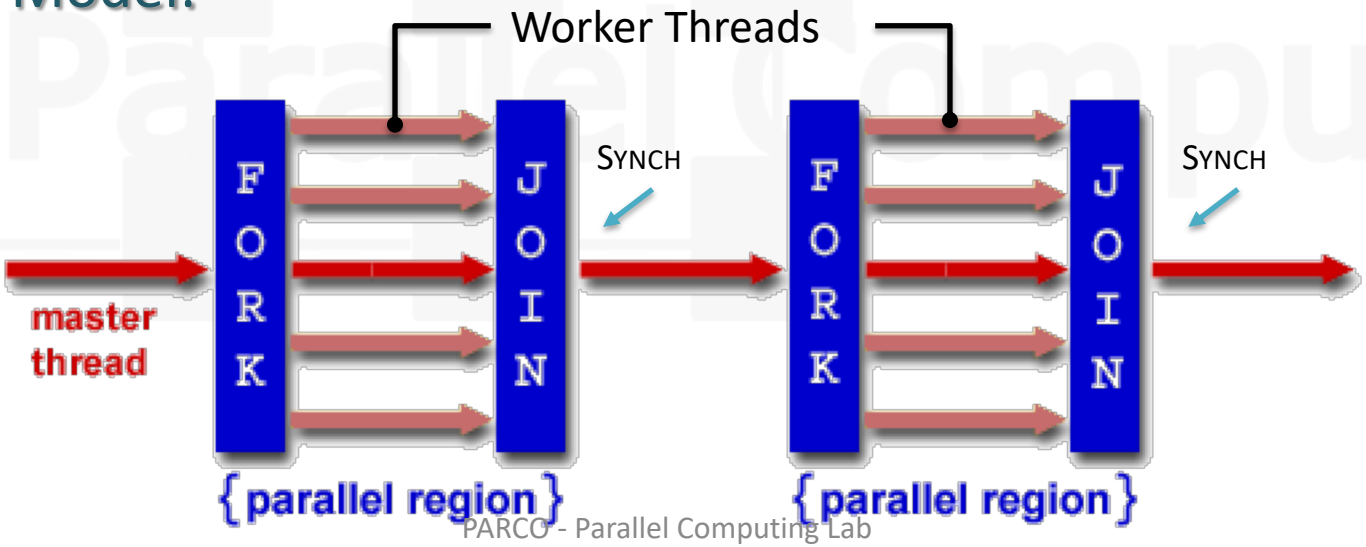
1) Identify the parallelism that sequential program contains (i.e. finding instructions or regions of code that may be executed concurrently by different processors)
2) Express the parallelism that has been identified by using OPENMP DIRECTIVES

- Modifications to the original source program are often needed in just a few places
- In general, the main effort in parallelizing a program with OpenMP lies in identifying the parallelism (step 1)

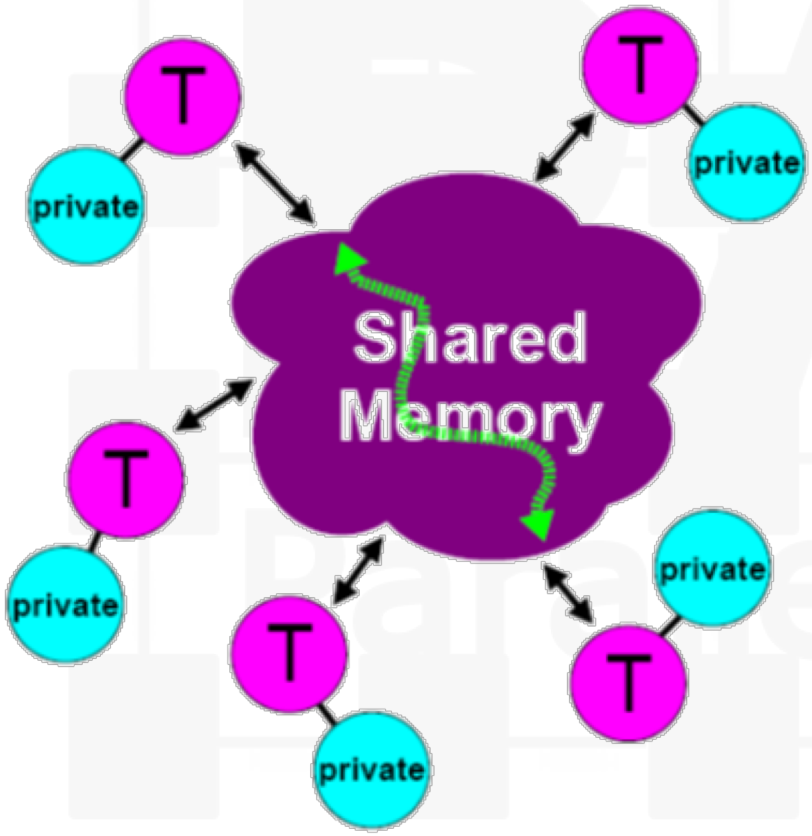
The OpenMP Execution Model

- Execution starts with one thread of control (*master thread*)
- Parallel regions fork off new threads on entry (*team threads*)
- Threads join back together at the end of the region (*synchronization*)

Fork-Join Model:



The OpenMP Memory Model



- All threads have access to the same globally shared memory
- Data can be *shared* or *private*
  - *Shared* data is accessible by all threads
  - *Private* data can only be accessed by the thread that owns it

The OpenMP Terminology

- A *parallel region* is a block of code executed by all threads simultaneously
- An OpenMP *team* is the group of threads that currently execute
- A *work-sharing construct* divides the execution of the current code region among the members of a team (i.e. it splits work among threads)

The OpenMP Syntax

- All OpenMP directives in C/C++ are indicated with a #pragma omp and are case sensitive

```
#pragma omp <directive> [clause [clause] …]
```

- A #pragma directive is the method specified by the C standard for providing additional information to the compiler
- Many OpenMP directives are followed by clauses to specify additional information
- The pragma usually applies only to the statement (or block of code) immediately following it

The PARALLEL Pragma (I)

- The PARALLEL pragma starts a parallel region
- It creates a team of N threads (where N is determined at runtime, usually from the number of CPU cores), all of which execute the next statement (or the next block, if the statement is enclosed in {…})
- After the statement, the threads join back into one

```
#pragma omp parallel
{
    // Code inside this region runs in parallel
    printf("Hello!\n");
}
```

- This example prints the text "Hello!" followed by a newline as

```
$ Hello!
$ Hello!
$ Hello!
$ Hello!
```

The PARALLEL Pragma (II)

- You can explicitly specify the number of threads to be created in the team using the NUM_THREADS clause:

```
#pragma omp parallel num_threads(<N>)
```

```
#pragma omp parallel num_threads(3)
{
    // Code inside this region runs in parallel
    printf("Hello!\n");
}
```

- E.g.:

```
$ Hello!
$ Hello!
$ Hello!
```

The FOR Pragma (I)

- The for directive splits the for-loop so that each thread in the current team handles a different portion of the loop

```
#pragma omp parallel for
for (int n = 0; n < 10; ++n)
    printf("%d ", n);
```

- It is a work-sharing construct, so it does not launch any new thread → it must be enclosed in a parallel region
- There is a shorthand to combine both these directives:

The FOR Pragma (II): an example

```
#pragma omp parallel for
{
    for (int i = 0; i < 1024; ++i)
        C[i] = A[i] + B[i];
}
```

Thread 0	Thread 1	Thread 2	Thread 3
$i \in [0-255]$	$i \in [256-511]$	$i \in [512-767]$	$i \in [768-1023]$
A[i]	A[i]	A[i]	A[i]
+	+	+	+
B[i]	B[i]	B[i]	B[i]
=	=	=	=
C[i]	C[i]	C[i]	C[i]

The FOR ORDERED Pragma

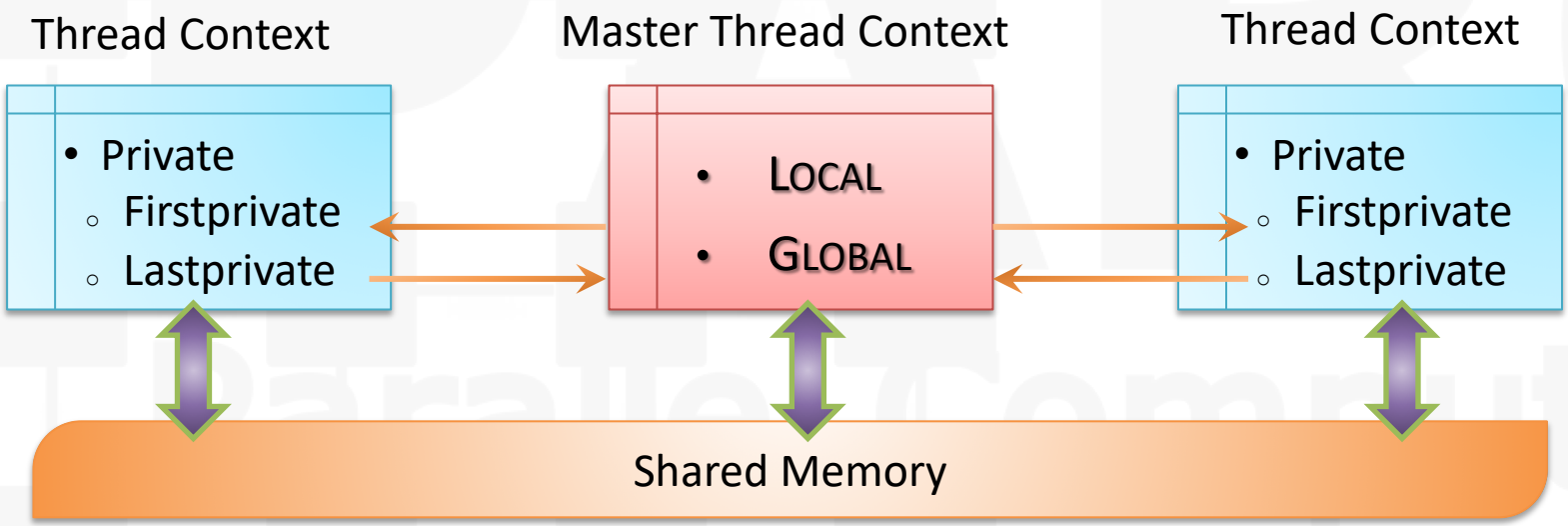
- The order in which loop iterations are executed is unspecified, it depends on runtime conditions
- However it is possible to force a block within a loop to be executed in the order in which loop iterations would be sequentially executed

```
#pragma omp for ordered
for (int n = 0; n < 10; ++n) {
    …
    #pragma omp ordered
    …
}
```

- Specifies that code under a parallelized for loop should be executed like a sequential loop
- There may be only one ordered block in the enclosing loop, which must contain the ordered clause

Data sharing (I)

Variable Scope:



Data sharing (II)
The SHARED Pragma

- The shared clause indicates that each variable in its list is *shared*, i.e. all threads access the same variable

```
#pragma omp parallel shared(<var1>, <var2>, … )
```

- A shared variable exists in only one memory location and all threads can read or write to that address
- It is the programmer's responsibility to ensure that multiple threads properly access SHARED variables (such as via CRITICAL sections)
- Because OpenMP is based upon the shared memory programming model, most variables are shared by default

Data sharing (III)
The PRIVATE Pragma

- In a parallel region it is possible to specify which variables are shared between threads and which are not
- The **private** clause indicates that each variable in its list is *private*, i.e. each thread has its own copy of it

```
#pragma omp parallel private(<var1>, <var2>, … )
```

- Private variables are undefined on entry and exit of the parallel region, i.e. a private copy is an uninitialized variable having the same name and the same type as the original variable
- The following are ***implicit private variables***:
  - Loop index variables
  - Stack variables in subroutines called from parallel regions

Data sharing (IV)
The FIRSTPRIVATE and LASTPRIVATE Pragma

- A private variable within a parallel region **does not** copy the value of the same variable inside and outside of the region

---

- To override this behavior, use the FIRSTPRIVATE clause instead

```
#pragma omp parallel firstprivate(<var1>, <var2>, … )
```

- All private copies of variables in the list are initialized with the value of the corresponding original variable before entering the region

---

- To preserve the last value of a private variable on exit of the parallel region, use the LASTPRIVATE clause

```
#pragma omp parallel lastprivate(<var1>, <var2>, … )
```

- All original variables corresponding to those in the list are updated after the end of the parallel region

RunTime Functions (I)

- Include omp Header

```
#include <omp.h>
```

- OpenMP C/C++ application program interface (API) for threads:
  - `int omp_get_max_threads()`

    Returns the maximum number of threads that would be available in a parallel region
  - `int omp_get_num_threads()`

    Returns the number of threads in the parallel region
  - `int omp_get_thread_num()`

    Returns the thread number of the thread executing within its thread team
  - `void omp_set_num_threads(int num_threads)`

    Sets the number of threads in subsequent parallel regions, unless overridden by a num_threads clause.

RunTime Functions (II)

- OpenMP C/C++ application program interface (API):
  - `double omp_get_wtime()`

    Returns a value in seconds of the time elapsed from some point.
  - `void omp_set_dynamic(int val)`

    Indicates that the number of threads available in subsequent parallel region can be adjusted by the run time
  - `int omp_in_parallel()`

    Returns nonzero if called from within a parallel region
  - `int omp_get_num_procs()`

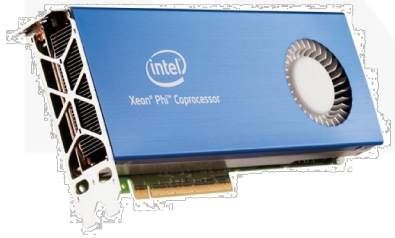
    Returns the number of processors that are available when the function is called

Examples

- To compile a program containing OpenMP directives you have to use gcc/g++ and add the –fopenmp flag:

```
g++ -fopenmp source.cpp -o out.x
```

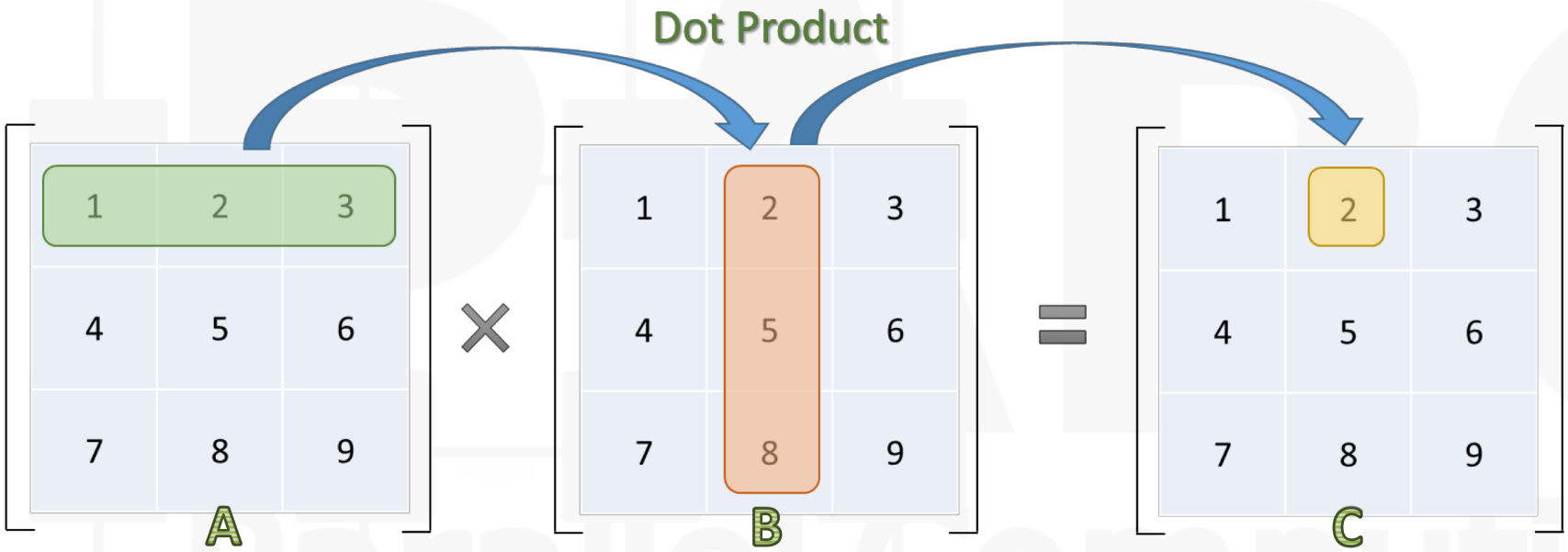
- gcc/g++ OpenMP Support:
  - GCC 4.2 implements version 2.5
  - GCC 4.4 implements version 3.0
  - GCC 4.7 implements version 3.1
  - GCC 4.9 implements version 4.0
  - GCC 5.x implements device Offloading



- **Examples:**
  - Hello World
  - Test Runtime Functions
  - Test PRIVATE / FIRSTPRIVATE / LASTPRIVATE Directives

Exercises (I)

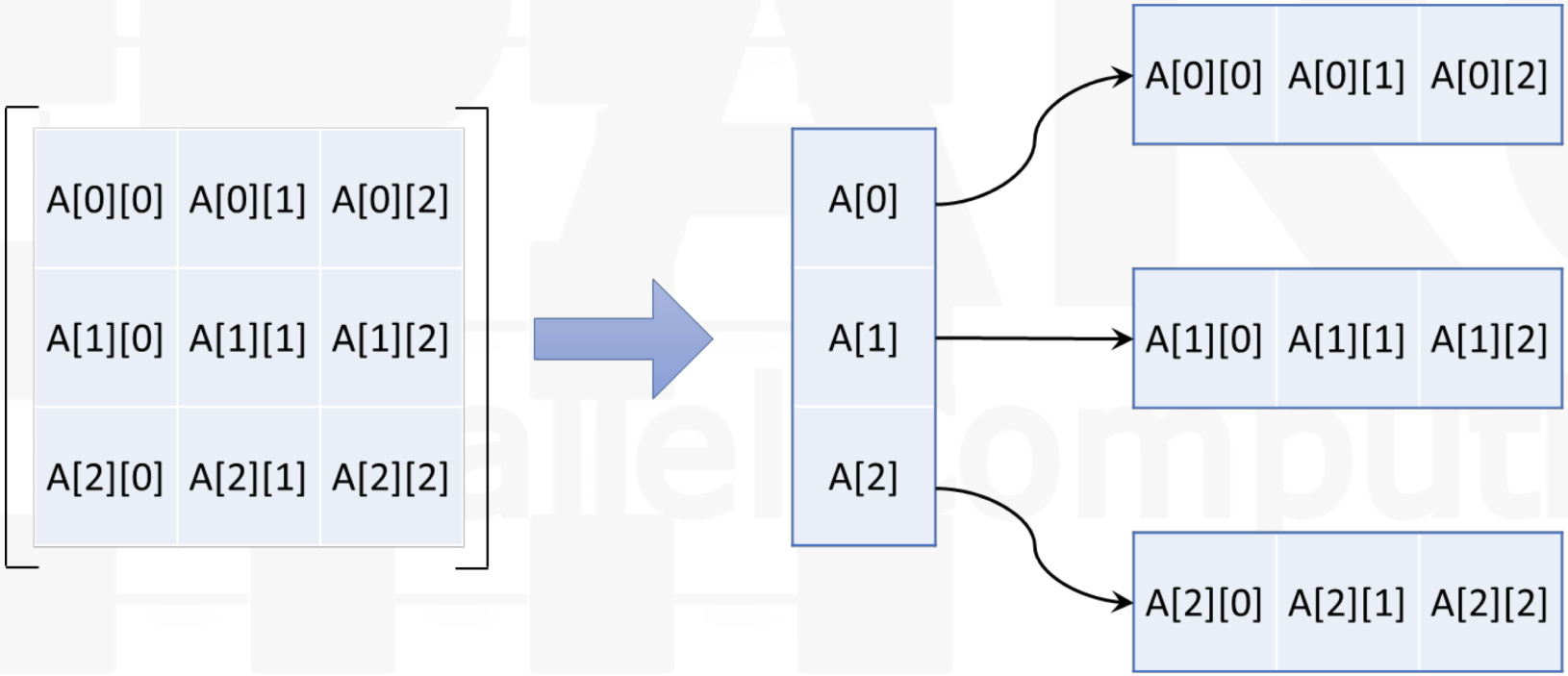
Matrix Multiplication




```
for row = 0 to N
    for col = 0 to N
        sum = 0
        for k = 0 to N
            sum += A[row][k] * B[k][col]
        C[row][col] = sum
    endfor
endfor
```

Exercises (I)

Matrix Multiplication



Exercises (II)

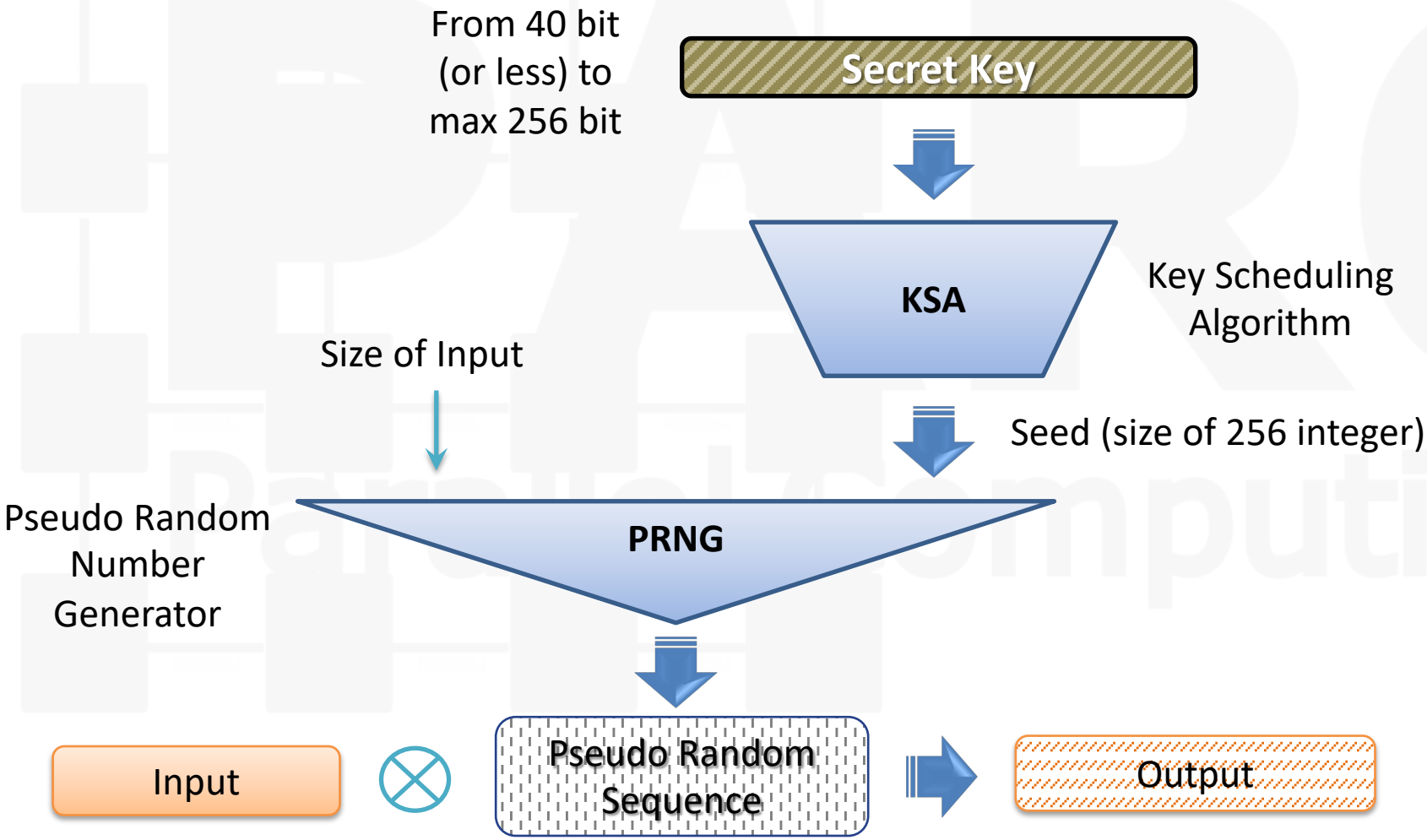
Simple exercises:

- Factorial(n)
  - n!
- Find:
  - Find two (given) consecutive numbers in an array

Little more complex:

- RC4 Cipher

RC4 Cipher


From 40 bit
(or less) to
max 256 bit
Secret Key
KSA
Key Scheduling
Algorithm
Size of Input
Seed (size of 256 integer)
Pseudo Random
Number
Generator
PRNG
Input
Pseudo Random
Sequence
Output

References

- Official OpenMP Specification (v4.0.0)
  www.openmp.org/mp-documents/OpenMP4.0.0.pdf
- OpenMP in Visual C++
  https://msdn.microsoft.com/en-us/library/tt15eb9t.aspx
- OpenMP Tutorial - Livermore Computing Center
  https://computing.llnl.gov/tutorials/openMP/
- Guide into OpenMP: Easy multithreading programming for C++
  http://bisqwit.iki.fi/story/howto/openmp/
- A “Hands-on” Introduction to OpenMP
  http://openmp.org/mp-documents/omp-hands-on-SC08.pdf