Scientific Programming

Dr. Vincent A. L. Boyer

Graduate Program in Systems Engineering (PISIS)
Facultad de Ingeniería Mecánica y Eléctrica
Universidad Autónoma de Nuevo León

E2023

Content

1. Introduction
2. Algorithm
3. Introduction to Programming
4. C++ Programming
5. Object Oriented Programming
6. CPLEX
7. Parallel Computing

Agenda

1. Introduction
2. Algorithm
3. Introduction to Programming
4. C++ Programming
5. Object Oriented Programming
6. CPLEX
7. Parallel Computing

Contact

Dr. Vincent A. L. Boyer
email: `vincent.boyer@uanl.edu.mx`
CIDET, 1st floor, Office 107

Evaluation Criteria

Criterion	Value
Homework	50%
Midterm Project	25%
Final Project	25%
Total	100%

Content

- I. Algorithm
- II. Introduction to Programming
- III. C++ Programming
- IV. Object Oriented Programming
- V. CPLEX
- VI. Parallel Computing

Bibliography

- Introduction to Algorithms. Charles E. Leiserson, Clifford Stein, Ronald Rivest, and Thomas H. Cormen. MIT Press. 1989
- Stroustrup, Bjarne. The C++ programming language. Pearson Education India. 1995.
- Schildt, Herbert. C++ from the Ground Up. McGraw-Hill. 2003.
- Soulié, Juan. C++ Language Tutorial. http://www.cplusplus.com/files/tutorial.pdf. 2007

Agenda

1 Introduction

2 Algorithm

3 Introduction to Programming

4 C++ Programming

5 Object Oriented Programming

6 CPLEX

7 Parallel Computing

Algorithm

Before there were computers, there were algorithms. But now that there are computers, there are even more algorithms, and algorithms lie at the heart of computing.

Algorithm 1 Make a Cup of Tea

```
Put teabag in cup
Fill kettle
Boil kettle
Pour water into cup
Add milk
Stir
Drink
```

Algorithm

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

For example, here is how we formally define the sorting problem:

Algorithm 2 Sorting Algorithm

Input: A sequence of n numbers $(a_1, a_2, ..., a_n)$.
Output: A permutation (reordering) $(a'_1, a'_2, ..., a'_n)$ of the input sequence such $a'_1 \leq a'_2 \leq ... \leq a'_n$.

Algorithm

For example, given the input sequence $(31, 41, 59, 26, 41, 58)$, a sorting algorithm returns as output the sequence $(26, 31, 41, 41, 58, 59)$. Such an input sequence is called an **instance** of the sorting problem. In general, an instance of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

An algorithm is said to be **correct** if, for every input instance, it halts with the correct output. We say that a correct algorithm solves the given computational problem. An incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer. Contrary to what you might expect, incorrect algorithms can sometimes be useful, if we can control their error rate.

Algorithm

Suppose computers were infinitely fast and computer memory was free. Would you have any reason to study algorithms? The answer is yes, if for no other reason than that you would still like to demonstrate that your solution method terminates and does so with the correct answer.

If computers were infinitely fast, any correct method for solving a problem would do. You would probably want your implementation to be within the bounds of good software engineering practice (for example, your implementation should be well designed and documented), but you would most often use whichever method was the easiest to implement.

Of course, computers may be fast, but they are not infinitely fast. And memory may be inexpensive, but it is not free. Computing time is therefore a bounded resource, and so is space in memory. You should use these resources wisely, and algorithms that are efficient in terms of time or space will help you do so.

Pseudocode

We shall typically describe algorithms as programs written in a **pseudocode** that is similar in many respects to C, C++, Java, Python, or Pascal. What separates pseudocode from "real" code is that in pseudocode, we employ whatever expressive method is most clear and concise to specify a given algorithm. Sometimes, the clearest method is English. Another difference between pseudocode and real code is that pseudocode is not typically concerned with issues of software engineering. Issues of data abstraction, modularity, and error handling are often ignored in order to convey the essence of the algorithm more concisely.

Computing $x \cdot y$

Computing $x \cdot y$

Algorithm 3 Computing $x \cdot y$

Input: Two positive integers x and y.

Output: $x \cdot y$.

```
$a \leftarrow 0$
for $i \leftarrow 1$ to $x$ do
    $a \leftarrow y + a$
return $a$
```

Computing $x \cdot y$

Algorithm 4 Computing $x \cdot y$

```
function MULTIPLY2(x,y)
    $a \leftarrow 0$
    for $i \leftarrow 1$ to $\min(x, y)$ do
        $a \leftarrow \max(x, y) + a$
    return $a$
```

Computing $x \cdot y$

Multiplication by duplation and mediation (around 1650 BC)

$$x \cdot y = \begin{cases} 0, & \text{if } x = 0; \\ \lfloor x/2 \rfloor \cdot (y + y), & \text{if } x \text{ is even}; \\ \lfloor x/2 \rfloor \cdot (y + y) + y, & \text{if } x \text{ is odd}. \end{cases}$$

Algorithm 5 Computing $x \cdot y$

```
function MULTIPLY3(x,y)
    if x=0 then
        return 0;
    else if x is even then
        return MULTIPLY2($\lfloor x/2 \rfloor$, $(y + y)$)
    else
        return MULTIPLY2($\lfloor x/2 \rfloor$, $(y + y)$)+y
```

Computing $x \cdot y$

Multiplication by duplation and mediation (around 1650 BC)

$$x \cdot y = \begin{cases} 0, & \textit{if } x = 0; \\ \lfloor x/2 \rfloor \cdot (y + y), & \textit{if } x \textit{ is even}; \\ \lfloor x/2 \rfloor \cdot (y + y) + y, & \textit{if } x \textit{ is odd}. \end{cases}$$

Algorithm 6 Computing $x \cdot y$

```
function MULTIPLY4(x,y)
    if x=0 then
        return 0;
    else if x is even then
        return MULTIPLY4($\lfloor x/2 \rfloor$, $(y + y)$)
    else
        return MULTIPLY4($\lfloor x/2 \rfloor$, $(y + y)$)+y
```

Computing $x \cdot y$

Multiplication by duplation and mediation (around 1650 BC)

$$x \cdot y = \begin{cases} 0, & \textit{if } x = 0; \\ \lfloor x/2 \rfloor \cdot (y + y), & \textit{if } x \textit{ is even}; \\ \lfloor x/2 \rfloor \cdot (y + y) + y, & \textit{if } x \textit{ is odd}. \end{cases}$$

Algorithm 7 Computing $x \cdot y$

```
function MULTIPLY4(x,y)
    $a \leftarrow 0$
    while $x > 0$ do
        if x is odd then
            $a \leftarrow a + y$
        $x \leftarrow \lfloor x/2 \rfloor$
        $y \leftarrow y + y$
    return a
```

x	y	a
		0
123	+456	=456
61	+912	=1368
30	~~1824~~	
15	+3648	=5016
7	+7296	=12312
3	+14592	=26904
1	+29184	=56088

Computing a^n

Computing a^n

Algorithm 8 Computing a^n

Input: A real a and a positive integer n.

Output: a^n.

```
$x \leftarrow 1$
for $i \leftarrow 1$ to $n$ do
    $x \leftarrow x \cdot a$
return $x$
```

Computing a^n

Divide and Conquer Method: $a^n = a^{\lfloor n/2 \rfloor} \cdot a^{\lceil n/2 \rceil}$

Computing a^n

Divide and Conquer Method: $a^n = a^{\lfloor n/2 \rfloor} \cdot a^{\lceil n/2 \rceil}$

Algorithm 9 Computing a^n

```
function DACPOWER(a,n)
    if n=0 then
        return 1
    if n=1 then
        return $a$
    else
        $x \leftarrow$ DACPOWER($a$, $\lfloor n/2 \rfloor$)
    if $n$ is even then
        return $x \cdot x$
    else
        return $x \cdot x \cdot a$
```

Sorting Algorithms

Algorithm 10 Insertion Sort

Input: An array A of n real value.

Output: The array A sorted in increasing order.

```
if $n = 1$ then
    STOP
for $j \leftarrow 2$ to $n$ do
    $key \leftarrow A[j]$
    $i \leftarrow j - 1$
    while $i > 0$ and $A[i] > key$ do
        $A[i+1] \leftarrow A[i]$
        $i \leftarrow i - 1$
    $A[i+1] \leftarrow key$
```

Sorting Algorithms

Homework Assignment: Investigate on the following other sorting algorithm.

- Merge Sort
- Bubble Sort
- Quick Sort
- Heap Sort
- Counting Sort

Pseudocode Conventions

- Indentation indicates block structure.
- The looping constructs while, for, and repeat-until and the if-else conditional construct have interpretations similar to those in C, C++, Java, Python, and Pascal.
- We use the keyword **to** when a **for** loop increments its loop counter in each iteration, and we use the keyword **downto** when a **for** loop decrements its loop counter. When the loop counter changes by an amount greater than 1, the amount of change follows the optional keyword **by**.
- The symbol // indicates that the remainder of the line is a comment.
- Variables (such as i, j, and key) are local to the given procedure. We shall not use global variables without explicit indication.

Pseudocode Conventions

- We access array elements by specifying the array name followed by the index in square brackets.
- We pass parameters to a procedure by value: the called procedure receives its own copy of the parameters, and if it assigns a value to a parameter, the change is not seen by the calling procedure.
- A return statement immediately transfers control back to the point of call in the calling procedure.
- The boolean operators “and” and “or” are short circuiting.
- The keyword error indicates that an error occurred because conditions were wrong for the procedure to have been called.

Exercises

Exercise 1: Illustrate the operation of insertion sort on the array $A = (31, 41, 59, 26, 41, 58)$.

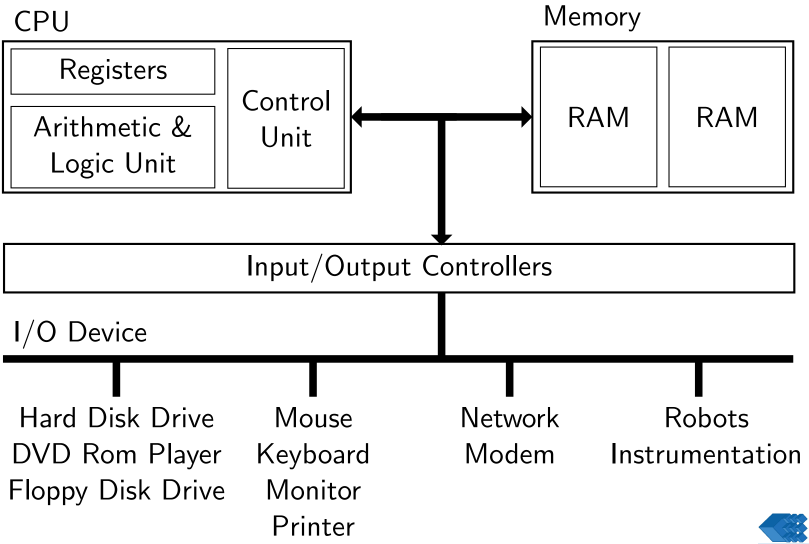
Exercise 2: Rewrite the insertion sort procedure to sort into nonincreasing instead of nondecreasing order.

Exercise 3: Write a pseudocode for linear search, which scans through an array A, looking for the value v.

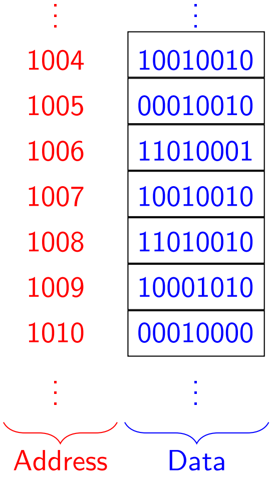
Agenda

1. Introduction
2. Algorithm
3. Introduction to Programming
4. C++ Programming
5. Object Oriented Programming
6. CPLEX
7. Parallel Computing

Computer Arquitecture



Memory Managment



Address	Data
⋮	⋮
1004	10010010
1005	00010010
1006	11010001
1007	10010010
1008	11010010
1009	10001010
1010	00010000
⋮	⋮

Programming Languages

Scripting/Interpreted Language:

Perl, Python, Shell, Java, ...

High/Middle Level Language:

C/C++, Fortran, Pascal, ...

Low Level Language:

Assembly Language.

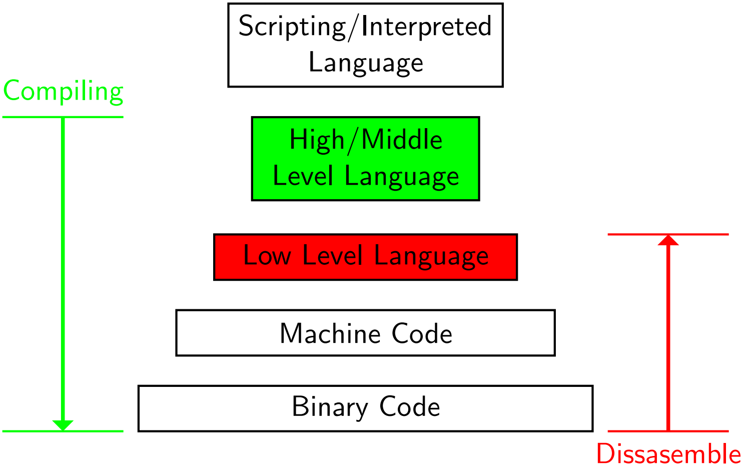
Machine Code:

Hexadecimal code read by the Operating System.

Binary Code:

Code read by hardware (not human readable).

Flow of Compilation



Agenda

1. Introduction
2. Algorithm
3. Introduction to Programming
4. C++ Programming
5. Object Oriented Programming
6. CPLEX
7. Parallel Computing

Hello World

```
  #include<iostream>
  using namespace std;

  int main(void){
// prints !!!Hello World!!!
    cout << "!!!Hello␣World!!!"<<endl;
    return 0;
  }
```

#include <iostream>

Tells the preprocessor to include the iostream standard file. This specific file (iostream) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

Hello World

```
  #include<iostream>
  using namespace std;

  int main(void){
// prints !!!Hello World!!!
    cout << "!!!Hello␣World!!!"<<endl;
    return 0;
  }
```

int main ()

This line corresponds to the beginning of the definition of the *main* function. The *main* function is the point by where all C++ programs start their execution, independently of its location within the source code.

Hello World

```
  #include<iostream>
  using namespace std;

  int main(void){
// prints !!!Hello World!!!
    cout << "!!!Hello␣World!!!"<<endl;
    return 0;
  }
```

cout << "Hello World!";

cout is the name of the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream (*cout*, which usually corresponds to the screen).

Hello World

```
  #include<iostream>
  using namespace std;

  int main(void){
// prints !!!Hello World!!!
    cout << "!!!Hello␣World!!!"<<endl;
    return 0;
  }
```

return 0;

The *return* statement causes the main function to finish. *return* may be followed by a return code (in our example is followed by the return code with a value of zero).

Hello Everyone

```
#include<iostream>
using namespace std;

int main(int argc, char* argv[]){

  for(int i=1;i<argc;i++)
    cout << "!!!Hello␣" << argv[i] << "␣!!!"
    <<endl;

  return 0;
}
```

Hello Everyone

```
#include<iostream>
using namespace std;

int main(int argc, char* argv[]){

  for(int i=1;i<argc;i++)
        cout << "!!!Hello␣" << argv[i] << "␣!!!" <<endl;
      return 0;
}
```

The *main* function parameters

argc Non-negative value representing the number of arguments passed to the program from the environment in which the program is run.

argv Pointer to the first element of an array of pointers to null-terminated multibyte strings that represent the arguments passed to the program from the execution environment (argv[0] through argv[argc-1]). The value of argv[argc] is guaranteed to be a null pointer.

Passing Arguments to the Executable

From command line

Open the console and go to the folder where your executable is saved.

- Windows: *MyProgram.exe Robert Alain Jean*
- OSX/Linux: ./*MyProgram Robert Alain Jean*

From your IDE

- Visual Studio: Go to Project → Properties → Configuration Properties → Debugging → Command Arguments.
- XCode: Go to Product → Scheme → Edit Scheme → Arguments.

Variables

Declaration of Variables

```
int a;
float mynumber;
```

Initialization of Variables

```
int a=0;
int b(0);
```

Assignment

```
int a=4;
int b;
b=a+5;
```

Data Types

Keyword	**Variable Type**	**Range**
char	Character	-128 to 127
int	integer	-2,147,483,648 to 2,147,483,647
short int	Short integer	-32,768 to 32,767
long int	Long integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned char	Unsigned character	0 to 255
unsigned int	Unsigned integer	0 to 4,294,967,295
unsigned short	Unsigned short integer	0 to 65,535
unsigned long	Unsigned long integer	0 to 18,446,744,073,709,551,615
float	Single-precision floating point	±3.4e ±38 (∼7 digits)
double	Double-precision floating point	±1.7e ±308 (∼15 digits)

Control Structure

Conditional Structure: *if* and *else*

```
if (x > 0)
  cout << "x␣is␣positive";
else if (x < 0)
  cout << "x␣is␣negative";
else
  cout << "x␣is␣0";
```

Control Structure

Conditional Structure: *switch*

```
int a = 10;
int b = 10;
int c = 20;
switch ( a ) {
case b:
  cout << "a==b";
  break;
case c:
  cout << "a==c";
  break;
default:
  cout << "a!=b␣and␣a!=c";
  break;
 }
```

Control Structure

Iteration Structure: *while*

```
int n;
cout << "Enter␣the␣starting␣number␣>␣";
cin >> n;
while (n>0) {
  cout << n << ",␣";
 --n;
}
cout << "FIRE!\n";
```

Control Structure

Iteration Structure: *do – while*

```
int n;
cout << "Enter␣the␣starting␣number␣>␣";
cin >> n;
do {
  cout << n << ",␣";
 --n;
}while (n>0);
cout << "FIRE!\n";
```

Control Structure

Iteration Structure: *for*

```
for (int n=10; n>0; n--) {
  cout << n << ",␣";
}
cout << "FIRE!\n";
```

Exercises (Control Structure)

Exercise 4: Write a program that asks the user to type the width and the height of a rectangle and then outputs to the screen the area and the perimeter of that rectangle.

Exercise 5: Write a program that asks the user to type 5 integers and writes the average of the 5 integers. This program can use only 2 variables.

Exercise 6: Write a program that asks the user to type the coordinate of 2 points, A and B (in a plane), and then writes the distance between A and B.

Exercise 7: Write a program that asks the user to type all the integers between 8 and 23 (both included) using a for loop.

Exercise 8: Same as exercise 4 but you must use a while.

Exercise 9: Write a program that asks the user to type 10 integers and writes the smallest value.

Exercises (Control Structure)

Exercise 10: Write a program that asks the user to type an integer N and compute u(N) defined with : $u(0)=1$; $u(1)=1$; $u(n+1)=u(n)+u(n-1)$.

Exercise 11: Write a program that asks the user to type the value of N and computes N! .

Exercise 12: Write a program that asks the user to type an integer N and that indicates if N is a prime number or not.

Exercise 13: Write a program that chooses a random number between 1 and 100. The user then tries to guess that number. The program will tell the user whether the hidden number is higher or lower than the number that was guessed. When the correct number is guessed, report back how many guesses it took to find the correct number. Then offer to play the game again.

Pointers

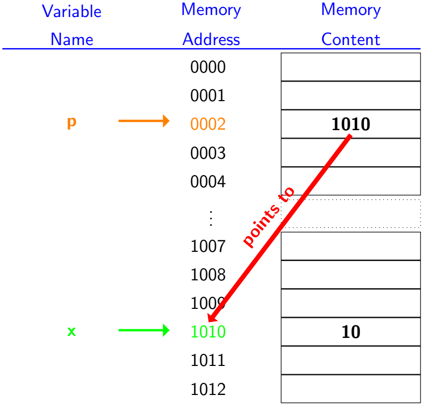
```
int x;       // A normal integer
int *p;      // A pointer to an integer
p = &x;      // Read it, "assign the address of x to p"
x=10;
cout<< *p <<"\n";
```

Definition

The variable that stores the address of another variable (like *p* in the previous example) is what in C++ is called a pointer. Pointers are a very powerful feature of the language that has many uses in lower level programming. For a C++ program, the memory of a computer is like a succession of memory cells, each one byte in size, and each with a unique address.

Pointers

```
int x;
int *p;
p = &x;
x=10;
cout<< *p <<"\n";
```



Arrays / Tables / Matrices

```
int array[8][8]; // Declares an array like a chessboard
for (int x = 0; x < 8; x++ ) {
  for ( y = 0; y < 8; y++ )
    array[x][y] = x * y; // Set each element to a value
}
cout<<"Array␣Indices:\n";
for (int x = 0; x < 8;x++ ) {
  for (int y = 0; y < 8; y++ )
    cout<<"["<<x<<"]["<<y<<"]="<< array[x][y] <<"␣";
    cout<<"\n";
}
```

Structure

```
struct database {
  int id_number;
  int age;
  float salary;
};
int main() {
  database employee;  //An employee variable that has
                      //modifiable variables inside it.
  employee.age = 22;
  employee.id_number = 1;
  employee.salary = 12000.21;
}
```

Operator *new* and *delete*

Dynamic Memory Allocation (1 Dimension)

```
int main() {
  int n, *pointer;
  cout << "Enter␣an␣integer\n";
  cin >> n;
  pointer = new int[n];
  cout << "Enter␣" << n << "␣integers\n";
  for ( int c = 0 ; c < n ; c++ )
    cin >> pointer[c];
  cout << "Elements␣entered␣by␣you␣are\n";
  for ( int c = 0 ; c < n ; c++ )
    cout << pointer[c] << endl;
  delete[] pointer;
  return 0;
}
```

Operator *new* and *delete*

Dynamic Memory Allocation (>1 Dimension)

```
int n=10, m=100,l=5;
int ***pointer;
pointer=new int**[n];
for ( int i = 0 ; i < n ; i++ ){
  pointer[i]=new int*[m];
  for ( int j = 0 ; j < m ; j++ ){
    pointer[i][j]=new int[l];
    for ( int k = 0 ; k < l ; k++ ){
      cout<<"Enter␣Value␣at␣Position␣("<<i<<","<<j<<","<<k<<")";
      cin>>pointer[i][j][k];
    }
  }
}

//Do some stuffs with pointer...

for ( int i = 0 ; i < n ; i++ ){
  for ( int j = 0 ; j < m ; j++ ){
    delete [] pointer[i][j];
  }
  delete [] pointer[i];
}
delete [] pointer;
```

Functions

Declaration Before the *main* Function

```
#include <iostream>
using namespace std;
int mult ( int x, int y )
{
  return x * y;
}
int main() {
  int x; int y;
  cout<<"Please␣input␣two␣numbers:␣";
  cin>> x >> y;
  cin.ignore();
  cout<<"The␣product␣is␣"<< mult ( x, y )<<"\n";
  cin.get();
}
```

Functions

Declaration After the *main* Function

```
#include <iostream>
using namespace std;
int mult ( int x, int y );
int main() {
  int x; int y;
  cout<<"Please␣input␣two␣numbers:␣";
  cin>> x >> y;
  cin.ignore();
  cout<<"The␣product␣is␣"<< mult ( x, y )<<"\n";
  cin.get();
}
int mult ( int x, int y )
{
  return x * y;
}
```

Functions

Declaration in Separate Files

MyFunctions.hpp

```
int mult ( int x, int y );
```

MyFunction.cpp

```
#include "MyFunctions.hpp"

int mult ( int x, int y ){
  return x * y;
}
```

MyCode.cpp

```
#include "MyFunctions.hpp"

int main() {
  int x; int y;
  cout<<"Please␣input␣two␣numbers:␣";
  cin>> x >> y;
  cin.ignore();
  cout<<"The␣product␣is␣"<< mult ( x, y )<<"\n";
  cin.get();
}
```

Exercises (Pointers and Functions)

Exercise 14: Write a program that asks the user to type the size N and the quantity K of tables to generate, then calls a function that will fill randomly K tables of size N with values (not only integers) in $[0, 1000]$. This function should return a pointer to an array of arrays containing all the generated tables.

Exercise 15: Modify the previous function such that it returns K tables of randomly sizes in $[10, N]$ (The program should check if the the value N typed by the user is greater to 10).

Exercise 16: Write a function that is taking a pointer to a table as a parameter and print its stored value on the screen. Add this function to the previous program to print the generated tables.

Exercises (Pointers and Functions)

Exercise 17: Add to the previous program a function that is taking a pointer to a table as a parameter and sorts the values of this table in decreasing order (Method of your choice). Test it with the K tables generated randomly.
Exercise 18: Modify the previous function so that it returns the time spent to sort the table (Use the function clock() from the library time.h). Print the average time spent to sort the K tables.

Input / Output

```cpp
#include <fstream>
#include <iostream>
#include <string>

using namespace std;

int main()
{
  string str;

  //Creates an instance of ofstream, and opens example.txt
  ofstream a_file ( "example.txt" );
  // Outputs to example.txt through a_file
  a_file<<"This␣text␣will␣now␣be␣inside␣of␣example.txt";
  // Close the file stream explicitly
  a_file.close();
  //Opens for reading the file
  ifstream b_file ( "example.txt" );
  //Reads one string from the file
  b_file>> str;
  //Should output 'this'
  cout<< str <<"\n";
  cin.get();    // wait for a keypress
  // b_file is closed implicitly here
}
```

Scripting

MyScript.sh (Unix)

```
#!/bin/sh
# This is a comment!
echo Hello World
for i in 1 2 3 4 5
do
  echo "number␣$i"
done
```

(chmod a+rx MyScript.sh)

MyScript.cmd (Windows)

```
@echo off
Rem This is a comment!
echo Hello World
for /l %%x in (1, 1, 100) do (
   echo number %%x
)
pause
```

Exercice (Input/Output)

Exercise 19: Write a program that can receive through *argv*[] parameters and excecutes the following actions according to the number of parameters:

- if the number of parameters is 3 (a string *FileName*, an integer *K*, and an integer *n*), a function *Generate* is called that generates *K* tables of *n* integers and then an other function *Save* is excecuted to write them in a file called *FileName*.
- if the number of parameter is 1 (a string *FileName*), a function *Read* is called that read *FileName* and save all the data of the tables. Then an another function *SORT* will order these tables in decreasing order using a sorting algorithm of your choice and display the average processing time.

You should use a structure to save the data of the tables.

Exercice (Input/Output)

Exercise 20: Modify the previous exercise, so that if the number of parameters is 2 (an integer K, and an integer n), the function *Generate* is called and then *SORT*.

Exercise 21: Modify the previous function *SORT* so that the result is written in a file named *SAN_K.txt* (use *ostringstream*), where:

- *SAN* is the name of the sorting algorithm;
- K is the total number of tables.

The result is saved in the format "*n processingtime*". If the file already exists, the function should append the new result on a different line.

Exercise 22: Write a script that run your program with different values of n (from 1000 to 100000) with $K = 10$ using at least 3 sorting algorithms of your choice. Use the resulting files to plot (Excel/R/Gnuplot/...) and compare the performance of these sorting algorithms.

Agenda

1. Introduction
2. Algorithm
3. Introduction to Programming
4. C++ Programming
5. Object Oriented Programming
6. CPLEX
7. Parallel Computing

Definitions

Class

A class is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.

Object

An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Definitions

Class Declaration

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
} object_names;
```

Definitions

Class Declaration

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
} object_names;
```

Class Identifier

The field *class_name* is a valid identifier for the class and *object_names* is an optional list of names for objects of this class.

Definitions

Class Declaration

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
} object_names;
```

Access Specifier

An access specifier modify the access rights that the members following them acquire. It is one of the following three keywords:

- *private* members of a class are accessible only from within other members of the same class or from their friends.
- *protected* members are accessible from members of their same class and from their friends, but also from members of their derived classes.
- *public* members are accessible from anywhere where the object is visible.

Definitions

Class Declaration

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
} object_names;
```

Members

The body of the declaration can contain *members*, that can be either data or function declarations. By default, all members of a class declared with the *class* keyword have private access for all its members. Therefore, any member that is declared before one other class specifier automatically has private access.

Class Declaration

Class Rectangle

```
class Rectangle {
  int width, length;
  public:
    void setvalues (int,int);
    int area (void);
} rect;
```

This declares a class (i.e., a type) called *Rectangle* and an object (i.e., a variable) of this class called *rect*. This class contains two data members of type int (member *width* and member *length*) with private access (because private is the default access level) and two member functions with public access (*setvalues*() and *area*()), of which for now we have only included their declaration (not their definition).

Class Declaration

```
#include <iostream>
using namespace std;

class Rectangle {
  int width, length;
  public:
    void setvalues (int,int);
    int area (void) {return width*length};
};

void Rectangle::setvalues(int a, int b){
  width = a;
  length = b;
}

int main(){
  Rectangle recta, rectb;
  recta.set_values (3,4);
  rectb.set_values (5,6);
  cout << "recta␣area:␣" << recta.area()<<endl;
  cout << "rectb␣area:␣" << rectb.area()<<endl;
  return 0;
}
```

Class Declaration

```cpp
#include <iostream>
using namespace std;

class Rectangle {
  int width, length;
  public:
    void setvalues (int,int);
    int area (void) {return width*length};
};

void Rectangle::setvalues(int a, int b){
  width = a;
  length = b;
}

int main(){
  Rectangle recta, rectb;
  recta.set_values (3,4);
  rectb.set_values (5,6);
  cout << "recta␣area:␣" << recta.area()<<endl;
  cout << "rectb␣area:␣" << rectb.area()<<endl;
  return 0;
}
```

Operator ::

This operator is used to define a member of a class from out side the class definition itself.

Member Function Definition

The definition of a member function can be included directly in the definition of the class (see *area*) or can be outside of the definition of the class (see *setvalues*).

Constructor

```cpp
#include <iostream>
using namespace std;

class Rectangle {
  int width, length;
  public:
    Rectangle (int,int);
    int area (void) {return width*length};
};

Rectangle::Rectangle(int a, int b){
  width = a;
  length = b;
}

int main(){
  Rectangle recta(3,4);
  Rectangle rectb(5,6);
  cout << "recta␣area:␣" << recta.area()<<endl;
  cout << "rectb␣area:␣" << rectb.area()<<endl;
  return 0;
}
```

Constructor

A class can include a special function called constructor, which is automatically called whenever a new object of this class is created. This constructor function must have the **same name as the class**, and cannot have any return type; not even void.

Constructor

```
#include <iostream>
using namespace std;

class Rectangle {
  int width, length;
  public:
    Rectangle();
    Rectangle(int,int);
    int area(void) {return width*length};
};

Rectangle::Rectangle(){
  width = 5;
  length = 5;
}

Rectangle::Rectangle(int a, int b){
  width = a;
  length = b;
}

int main(){
  Rectangle recta(3,4);
  Rectangle rectb;
  cout << "recta␣area:␣" << recta.area()<<endl;
  cout << "rectb␣area:␣" << rectb.area()<<endl;
  return 0;
}
```

Constructor Overloading

Like any other function, a constructor can also be overloaded with more than one function that have the same name but different types or number of parameters. Remember that for overloaded functions the compiler will call the one whose parameters match the arguments used in the function call.

Destructor

```
#include <iostream>
using namespace std;

class Rectangle {
  int *width, *length;
  public:
    Rectangle (int,int);
    ~Rectangle();
    int area (void) {return *width * *length};
};

Rectangle::Rectangle(int a, int b){
  width=new int;
  length=new int;
  *width = a;
  *length = b;
}

Rectangle::~Rectangle{
  delete width;
  delete length;
}

int main(){
  Rectangle recta(3,4);
  Rectangle rectb(5,6);
  cout << "recta␣area:␣" << recta.area()<<endl;
  cout << "rectb␣area:␣" << rectb.area()<<endl;
  return 0;
}
```

Destructor

The destructor fulfills the opposite functionality. It is automatically called when an object is destroyed, either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using the operator delete. The destructor must have the **same name as the class, but preceded with a tilde sign (∼)** and it must also return no value.

Exercises

Exercise 23: Write a class *CTable* that holds all the required information to store a table. This class should contain a constructor where the user only specify the size of the table and then it is filled randomly, and an other constructor that read the data of the table from a file. This class should also contain a member function *Print*() to display the data in the stored table, a member function *Save*(*string*) to save the table in a file, a member function *Sort*() to sort the data in the table in decreasing order, and a member function *Average*() to return the average value of the data stored in the table. Do not forgot to define the destructor to free the memory.

Friendship

```
#include <iostream>
using namespace std;

class Rectangle {
  int width, length;
  public:
    Rectangle (int,int);
    int area (void) {return width * length};
    friend Rectangle duplicate (Rectangle);
};

Rectangle::Rectangle(int a, int b){
  width = a;
  length = b;
}

Rectangle duplicate (Rectangle rectparam) {
  Rectangle rectres(rectparam.width,rectparam.length);
  rectres.width *=2;
  rectres.length *=2;
  return rectres;
}

int main(){
  Rectangle recta(3,4);
  Rectangle rectb=duplicate(recta);
  cout << "recta␣area:␣" << recta.area()<<endl;
  cout << "rectb␣area:␣" << rectb.area()<<endl;
  return 0;
}
```

Friend Function

An external function declared as friend of a class allows this function to have access to the private and protected members of this class. Such a function is declared within the class, and is preceded with the keyword *friend*.

Friendship

```
#include <iostream>
using namespace std;

class Square;

class Rectangle {
  int width, length;
  public:
    Rectangle (int,int);
    Rectangle (Square a);
    int area (void) {return width * length};
};

class Square {
  private:
    int side;
  public:
    Square (int a) {side=a;}
    friend class Rectangle;
};

Rectangle::Rectangle(int a, int b){
  width = a;
  length = b;
}
```

```
Rectangle:: Rectangle (Square a); {
  width=a.side;
  length=a.side;
}

int main(){
  Square sqr(5);
  Rectangle rect(sqr);
  cout << "rect␣area:␣" << rect.area()<<en
  return 0;
}
```

Friend Class

A friend class of another class grant to that first class access to the protected and private members of the second one.

Inheritance

```
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, length;
  public:
    Polygon (int a, int b){
      width=a;
      length=b;
      cout<<"Polygon␣Constructor"<<endl;
    }
};

class Rectangle: public Polygon {
  public:
    Rectangle(int a, int b) : Polygon(a,b){
      cout<<"Rectangle␣Constructor"<<endl;
    }
    int area () {return width * length};
};

class Triangle: public Polygon {
  public:
    Triangle(int a, int b): Polygon(a,b){
      cout<<"Triangle␣Constructor"<<endl;
    }
    int area () {return width * length / 2};
};
```

```
int main () {
  Rectangle rect(4,5);
  Triangle trgl(4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

Inheritance

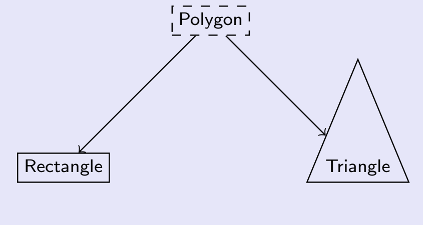
Inheritance allows to create classes which are derived from other classes, so that they automatically include some of its ”parent's” members, plus its own.

Inheritance

```
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, length;
  public:
    Polygon (int a, int b){
      width=a;
      length=b;
      cout<<"Polygon␣Constructor"<<endl;
    }
};

class Rectangle: public Polygon {
  public:
    Rectangle(int a, int b) : Polygon(a,b){
      cout<<"Rectangle␣Constructor"<<endl;
    }
    int area () {return width * length};
};

class Triangle: public Polygon {
  public:
    Triangle(int a, int b): Polygon(a,b){
      cout<<"Triangle␣Constructor"<<endl;
    }
    int area () {return width * length / 2};
};
```

```
int main () {
  Rectangle rect(4,5);
  Triangle trgl(4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

Inheritance



Polymorphism

```cpp
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, length;
  public:
    Polygon (int a, int b){
      width=a;
      length=b;
    }
    void Print (){
      cout<<width<<"␣-␣"<<length<<endl;
    }
};

class Rectangle: public Polygon {
  public:
    Rectangle(int a, int b) : Polygon(a,b){}
    int area () {return width * length};
};

class Triangle: public Polygon {
  public:
    Triangle(int a, int b): Polygon(a,b){}
    int area () {return width * length / 2};
};
```

```cpp
int main () {
  Rectangle rect(4,5);
  Triangle trgl(3,2);
  Polygon *P1=&rect;
  Polygon *P2=&trgl;
  P1->Print();
  P2->Print();
  return 0;
}
```

Polymorphism

A pointer to a derived class is type-compatible with a pointer to its base class. Polymorphism brings Object Oriented Methodologies to its full potential.

Exercises

Exercise 24: Write a class *SortedTable* derived from the previous class *CTable*. This class should contain a table that is alway sorted in decreasing order. Two constructors should be defined: a default one that initializes an empty table (size 0) and another one that copies the data from an object of *CTable*. This class should also contain a member function $insert(double)$ to insert a new value in the table in decreasing order and a member function $erase(double)$ that erase a value from the table. In both of these functions, the size of the table should be dynamically modified when required.

Exercise 25: Write a class *SortedList* derived from the class $list < double >$. This class should contain a member function $insert(double)$ to insert a new value in the list in decreasing order and a member function $erase(double)$ to erase a value from the list. Compare the processing time required to insert and erase an element with the one of the class *SortedTable*.

Virtual Member

```
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, length;
  public:
    Polygon (int a, int b){
      width=a;
      length=b;
    }
    virtual int area (){
      return 0;
    }
};

class Rectangle: public Polygon {
  public:
    Rectangle(int a, int b) : Polygon(a,b){}
    int area () {return width * length};
};

class Triangle: public Polygon {
  public:
    Triangle(int a, int b): Polygon(a,b){}
    int area () {return width * length / 2};
};
```

```
int main () {
  Rectangle rect(4,5);
  Triangle trgl(4,5);
  Polygon poly(4,5);
  Polygon *P1=&rect;
  Polygon *P2=&trgl;
  Polygon *P3=&poly;
  cout<<P1->area()<<endl;
  cout<<P2->area()<<endl;
  cout<<P3->area()<<endl;
  return 0;
}
```

Virtual Member

A member of a class that can be redefined in its derived classes is known as a virtual member. A class that declares or inherits a virtual function is called a *polymorphic* class.

Abstract Class

```
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, length;
  public:
    Polygon (int a, int b){
      width=a;
      length=b;
    }
    virtual int area ()=0;
};

class Rectangle: public Polygon {
  public:
    Rectangle(int a, int b) : Polygon(a,b){}
    int area () {return width * length};
};

class Triangle: public Polygon {
  public:
    Triangle(int a, int b): Polygon(a,b){}
    int area () {return width * length / 2};
};
```

```
int main () {
  Rectangle rect(4,5);
  Triangle trgl(4,5);
  Polygon *P1=&rect;
  Polygon *P2=&trgl;
  cout<<P1->area()<<endl;
  cout<<P2->area()<<endl;
  return 0;
}
```

Abstract Class

If a class contains at least one *pure* virtual member, it is called an abstract class. A derived class should define all the *pure* abstract members.

Exercise

Exercise 26: Write an abstract class *SortedContenair* that defined four pure abstract functions: *insert*(*double*) to insert a new value in decreasing order in the container; *erase*(*double*) to erase a value from the container; *Print*() to display the values contained in the container; and *check*() that returns true if the the values in the container are sorted correctly and false otherwise. From this class, rewrite the two previous classes *SortedTable* and *SortedList* as derived classes of *SortedContainer*. Design an experimentation protocol to validate these two classes and to show their advantages and disadvantages.

Agenda

1 Introduction

2 Algorithm

3 Introduction to Programming

4 C++ Programming

5 Object Oriented Programming

6 CPLEX

7 Parallel Computing

Tutorial

IBM Knowledge Center: https://www.ibm.com/support/knowledgecente
en/SSSA5P_12.6.3/ilog.odms.cplex.help/CPLEX/GettingStarted/top
tutorials/Cplusplus/cpp_synopsis.html

CPLEX Objects

IloEnv: CPLEX environment

The environment object provides (among other things) optimized memory management for objects of Concert Technology classes. It is the first object created in any Concert Technology application.

IloModel: Creating a Model

Objects of the class IloModel are used to define a complete optimization model that can later be extracted to an IloCplex object. The environment is passed as an argument to the constructor.

IloCplex: Solving the Model

The IloCplex object is used to extract the model to be solved. It also provides information whether it found the optimal solution or only a feasible solution, whether it proved the model to be unbounded or infeasible, or whether nothing at all has been proved at this point.

CPLEX Objects

IloNumVar: Modelling Variables

An instance of this class represents a numeric variable in a model. A numeric variable may be either an integer variable or a floating-point variable. It also has a lower and upper bound.

IloRange: Constraints

This class models a constraint of the form: *lowerBound* $<=$ *expression* $<=$ *upperBound*. After you create a constraint, such as an instance of IloRange, you must explicitly add it to the model in order for it to be taken into account.

IloObjective: The Objective Function

An objective consists of its sense (specifying whether it is a minimization or maximization) and an expression. The expression may be a constant, a numeric expression or a multiple criteria expression.

Building and Solving a Small LP Model

$$
\begin{array}{llclclcll}
Max & x_1 & + & 2x_2 & + & 3x_3 & & & \\
s.t. & -x_1 & + & x_2 & + & x_3 & \leq & 20 & (1) \\
 & x1 & - & 3x_2 & + & x_3 & \leq & 30 & (2) \\
 & x_1 & & & & & \leq & 40 & (3) \\
 & \multicolumn{5}{l}{x_1 \geq 0,\ x_2 \geq 0,\ x_3 \geq 0} & & &
\end{array}
$$

Building and Solving a Small LP Model

The Libraries

```
#include <ilcplex/ilocplex.h>

using namespace std;
```

Building and Solving a Small LP Model

The Libraries

```
#include <ilcplex/ilocplex.h>

using namespace std;
```

Defining the environment, the model, and the solver

```
IloEnv env;
IloModel model(env);
IloCplex cplex(model);
```

Building and Solving a Small LP Model

The Libraries

```
#include <ilcplex/ilocplex.h>

using namespace std;
```

Defining the environment, the model, and the solver

```
IloEnv env;
IloModel model(env);
IloCplex cplex(model);
```

Defining the Variables

```
IloNumVarArray vars(env);
vars.add(IloNumVar(env, 0.0, 40.0));
vars.add(IloNumVar(env));
vars.add(IloNumVar(env));
```

Building and Solving a Small LP Model

Defining the Constraints

```
IloRangeArray con(env);
con.add(- vars[0] + vars[1] + vars[2] <= 20);
con.add( vars[0] - 3 * vars[1] + vars[2] <= 30);
model.add(con);
```

Building and Solving a Small LP Model

Defining the Constraints

```
IloRangeArray con(env);
con.add( - vars[0] + vars[1] + vars[2] <= 20);
con.add( vars[0] - 3 * vars[1] + vars[2] <= 30);
model.add(con);
```

Solver Parameters

```
cplex.setParam(IloCplex::TiLim, 100.000);
cplex.setParam(IloCplex::Threads, 1);
cplex.setParam(IloCplex::EpGap, 0.0);
cplex.setParam(IloCplex::EpAGap, 0.0);
```

Building and Solving a Small LP Model

Defining the Constraints

```
IloRangeArray con(env);
con.add( - vars[0] + vars[1] + vars[2] <= 20);
con.add( vars[0] - 3 * vars[1] + vars[2] <= 30);
model.add(con);
```

Solver Parameters

```
cplex.setParam(IloCplex::TiLim, 100.000);
cplex.setParam(IloCplex::Threads, 1);
cplex.setParam(IloCplex::EpGap, 0.0);
cplex.setParam(IloCplex::EpAGap, 0.0);
```

Solving the Model

```
cplex.solve();
```

Building and Solving a Small LP Model

Getting the Solution

```
if (cplex.getStatus() == IloAlgorithm::Optimal) {
  IloNumArray vals(env);
  cplex.getValues(vals, vars);
  env.out() << "Values␣=␣" << vals << endl;
}
```

Exercise

Exercise 27: Implement in C++/CPLEX a model for the set covering problem and provide in a table the solution returned by CPLEX and the processing time for at least 10 different instances from the OR-Library. You should specify the parameters used in your experiment (time limit, gap, number of threads, ...).

Agenda

1. Introduction
2. Algorithm
3. Introduction to Programming
4. C++ Programming
5. Object Oriented Programming
6. CPLEX
7. Parallel Computing

Introduction

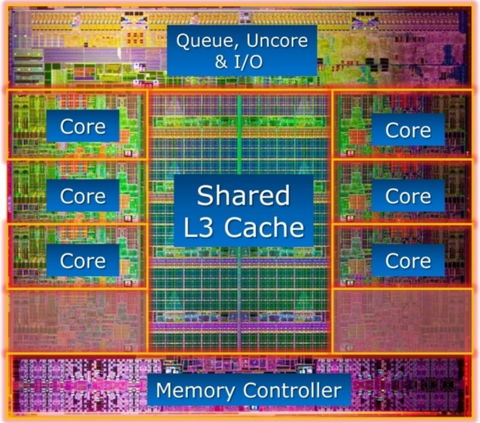
Why Parallel Computing?

- To solve larger problems (by distributing the memory requirement);
- To solve problem faster (by distributing the tasks).

Parallel Computing Examples

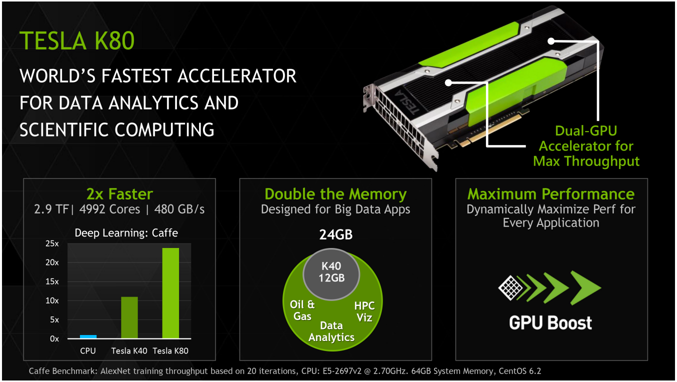
- Client and server on Internet;
- Search in a data base;
- Weather prediction;
- Astrophysics;
- Finite element simulation;
- Deep learning;
- Genetic engineering.

Parallel Computers: Multi-core Processor



Intel i7 Architecture

Parallel Computers: GPU



TESLA K80

WORLD'S FASTEST ACCELERATOR FOR DATA ANALYTICS AND SCIENTIFIC COMPUTING

Dual-GPU Accelerator for Max Throughput

2x Faster
2.9 TF| 4992 Cores | 480 GB/s

Deep Learning: Caffe

Double the Memory
Designed for Big Data Apps

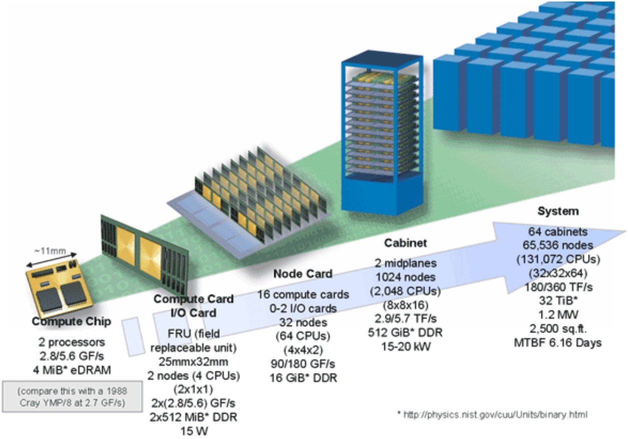
24GB

Maximum Performance
Dynamically Maximize Perf for Every Application

Caffe Benchmark: AlexNet training throughput based on 20 iterations, CPU: E5-2697v2 @ 2.70GHz. 64GB System Memory, CentOS 6.2

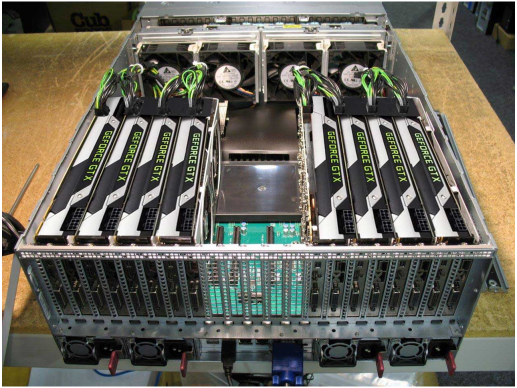
Nvidia Tesla T80

Parallel Computers: Cluster



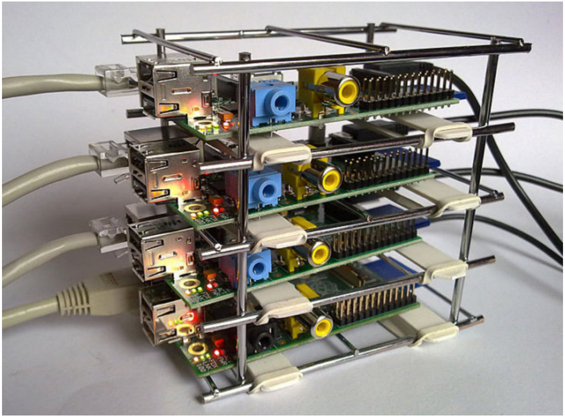
IBM Blue Gene

Parallel Computers: Cluster



Nvidia Cluster

Parallel Computers: Cluster

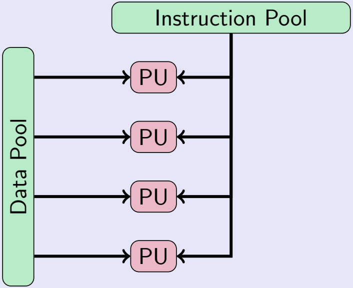


Rasberry Pi Cluster

Architecture

Single Instruction Multiple Data (SIMD)

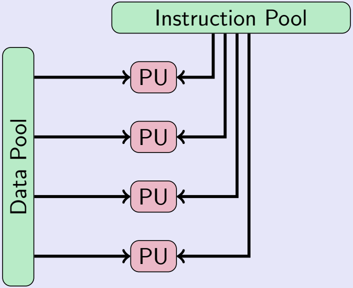
A single instruction stream is broadcasted to multiple processors, each having its own data stream (used in GPUs).



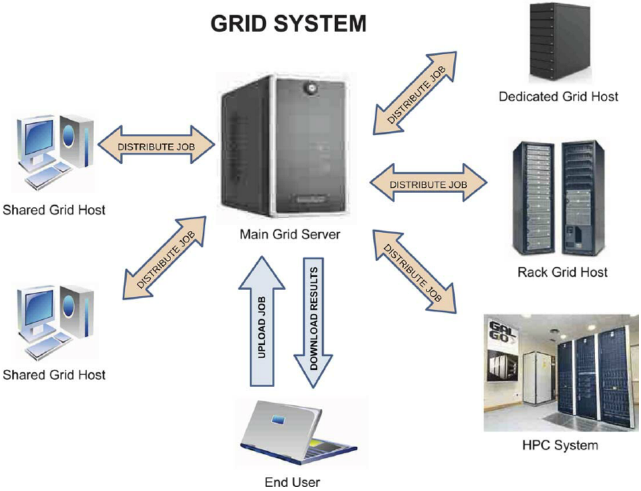
Architecture

Multiple Instructions Multiple Data (MIMD)

Each processor has its own instruction stream and input data (used in CPUs) with shared or distributed memory.



Grid


GRID SYSTEM
Dedicated Grid Host
DISTRIBUTE JOB
Shared Grid Host
DISTRIBUTE JOB
Main Grid Server
DISTRIBUTE JOB
Rack Grid Host
DISTRIBUTE JOB
DISTRIBUTE JOB
Shared Grid Host
UPLOAD JOB
DOWNLOAD RESULTS
HPC System
End User

C++ Parallel Programming Libraries

POSIX

POSIX threads, or Pthreads, refers to the tandardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard for UNIX Systems. Most hardware vendors offer Pthreads in addition to their proprietary Application Programming Interface (API).

OpenMP

OpenMP (Open Multi-Processing) supports multi-platform shared memory multiprocessing programming, jointly developed by a group of major computer hardware and software vendors. OpenMP is much higher level than Pthreads.

C++ Parallel Programming Libraries

MPI/OpenMPI

MPI (Message Passing Interface) target both distributed and shared memory system. It is a message passing standard designed to function on a wide variety of parallel computing architecture.

C++ Parallel Programming Libraries

CUDA

CUDA is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). It is compatible only with CUDA-Enabled GPUs (i.e. most Nvidia GPUs).

OpenCL

Open Computing Language (OpenCL) is a low-level API for heterogeneous platforms such as CPUs, GPUs, FGPAs, etc. It supports most of the recent CPUs and GPUs on the market.