Introduction to parallel programming

Parallel Computing is a type of computation where many calculations are performed at the same time.

Basic Principle: computations can be divided into smaller subproblems, each of which can be solved simultaneously.

History: Parallel programming has existed since the early days of computers. IBM built some of the first commercial parallel computers in the 20th century.

“For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers.” – Gene Amdahl, 1967

At the beginning of the 21st century, processor frequency scaling hit the power wall, which means the power required for the processer increases nonlinearly.

So, processor vendors began selling multiple CPU cores on the same processor chip, each capacle of executing separate instruction streams.

Parallel programming is much harder than sequential programming,

Separating sequential computations into parallel sub computations can be challenging, or even impossible.

Ensuring program correctness is more difficult, due to new types of errors.

Parallel programming vs concurrent programming

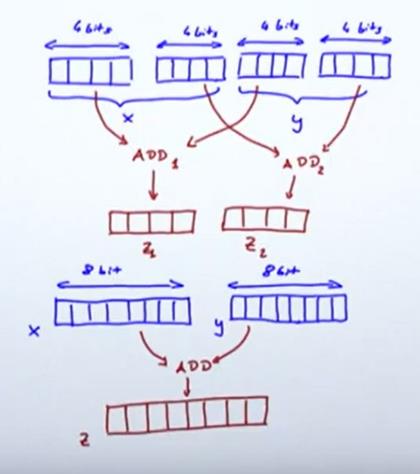
They are very closely related, but different.

A parallel program uses parallel hardware to execute computations more quickly, with efficiency as its main concern. Parallel programming solves the problems of dividing problems into subproblems, and figuring the optimal use of parallel hardware

A concurrent program is one that may or may not execute multiple executions at the same time. This improves modularity, responsiveness to events, and maintainability. Concurrent programming solves problems such as when can an execution start? And when and how can two concurrent executions share information. And how do computations manage access to shared resources?

Parallel programming is mainly concerned with answering algorithmic problems such as applications of matrix multiplication, data processing, computer graphics rendering, or simulations of fluid movement.

Concurrent programming is targeted at asynchronous applications such as web servers, user interfaces, or databases.

The biggest concern with parallel programming is the speed of the programs, whereas with concurrent programming, the biggest concern is with convenience.

Parallelism granularity levels

Bit level parallellism:

Some microproccessors have historitically had a 4 bit word size. Number ranges that did not fit into the 4 bits, had to be represented by multiple words. This meant that variables in the 8 bit range have to be represneted by 8 bits, that is 2 words. Adding these two varaibesl together required two seperare add instructiosn. Each operating on 4 bit words. Word size subsequently increased to 8 bit, thene 16 bits, then eventually 32 bits. Eeachh time this hapepned, several instructions could be replaced by a single one. This meant that processing more data in the same time interval, in event, preformance was improved by parallelizing operations on the bit level.

Instruction level parallelism:

This indicates executing different instructions from the same instruction stream in parallel. This can be done if both of the instructions do not depend on each other.

In this example,

Var a = 1 + 2

Var b = 4 + 5

Var c = a + b

There is no reason for a and b to not be computed at the same time because they are completely independent from each other. However, computing variable c can not be done in parallel with a or b because it depends on both of them.

Task level parallelism:

Task level parallelism is the task of executing entirely separate instruction streams in parallel. Bit level and instruction level parallelism may be implemented into the processor itself. However, task level parallelism is usually achieved through the support of software.

A multi-core processor is a processor that contains multiple processing cores on a single chip. Processor vendors such as intel, oracle, IBM all produce different models of multi-core processors.

A symmetric multiprocessor or SMP is a computer system with multiple identical processors that share memory and connect to it via bus. Here, multiple execution units are not on the same chip. An SMP may contain multi-core processors. A graphics processing unit is a form of coprocessor originally intended for graphics processing but can execute a program when this is requested by the host processor. Field programmable gate arrays (FPGA) is another form of a coprocessor which can manipulate itself for a given task. This can be advantageous in the form of improved performance when optimized for a specific application. Finally, computer clusters are groups of connected computers with no commonly shared memory.

(<https://www.youtube.com/watch?v=RNVIcm8-6RE>)

Introduction Part 2

**How To Use Raspberry Pi**

Before we begin setting up a raspberry pi, these are the items that we will need:

The Pi: Raspberry Pi 3 Model B+ ($40 at Amazon)

microSD card: SanDisk Ultra 32GB microSD card ($8 at Amazon)

microSD card reader: SanDisk Mobile Mate microSD card reader ($13 at Amazon)

Power supply: CanaKit 5V Raspberry Pi Power supply ($10 at Amazon)

USB keyboard: AmazonBasics Wired Keyboard ($14 at Amazon)

USB mouse: Logitech B100 ($10 at Amazon)

You'll also need a monitor or TV that accepts either HDMI or composite video input. HDMI works best, but composite video is workable. Many Raspberry Pi projects use an internet connection, so you'll also want a Wi-Fi dongle or ethernet cable.

**How to reformat your microSD card:**

Insert your microSD card into the USB card reader.

Connect the card reader to your computer.

Download SD Formatter 5.0.1.

Double-click on Install SD Card Formatter 5.0.1.mpkg in your downloads folder in your Dock to install SD Formatter 5.0.

Follow the instructions in the installation window.

Click the Launchpad icon in your Dock. It looks like a silver rocket ship.

Find the SD Formatter 5.0.1 app.

To move between Launchpad windows, click the Next Page icons at the bottom center of the screen, or swipe to the right or left with your trackpad or Magic Mouse.

Click on the SD Formatter 5.0.1 app to open it. A formatting window will appear on your desktop.

Under Select Card select your microSD card from the dropdown menu.

Click Format in the bottom right corner.

When the reformat is complete, you will get a notification window. Select OK to close the window. Your microSD card is now ready to install the operating system to the Raspberry Pi

**How to Download NOOBS onto the microSD card**

Download the ZIP file of NOOBS Version 3.0.0. It is a large file and will take a while to complete. You will want Raspbian, so do not download NOOBS Lite.

Double-click on the NOOBS file from the Downloads folder in your Dock to open it.

Select the first file inside the NOOBS folder.

Scroll down and Shift + left-click on the last file in the NOOBS folder.

Drag and drop all selected NOOBS files into the SD card icon on your desktop. You don't have to open the SD card drive.

Right-click on the SD card icon.

Select "Eject [SD Card Name]".

Remove the card reader from your computer.

Remove the microSD card from the card reader.

Now that NOOBS is loaded onto your microSD card, you're ready to set up your Raspberry Pi.

**Set up your Raspberry Pi**

Insert the microSD card into the card slot on the underside of the Raspberry Pi.

Plug the USB keyboard into one of the USB ports.

Plug the USB mouse into one of the USB ports

Alternatively, connect the Bluetooth adapter into one of the USB ports.

Turn on your monitor or TV set and make sure it is set to the proper input (e.g. HDMI 1 or Component)

Plug the HDMI or video component cable into the monitor or TV set.

Connect the other end of the cable into the Raspberry Pi.

Connect an ethernet cable to your router if you plan to connect to the Internet.

Connect the other end of the cable to your Raspberry Pi.

Alternately, connect the Wi-Fi adapter to the Raspberry Pi.

Connect the power supply to the Raspberry Pi.

Plug the power supply into the power outlet. This will turn on and boot up Raspberry Pi. A power indicator light will begin to glow, letting you know that you are connected.

A start screen should appear on the monitor or TV you're using.

**Download the Raspbian operating system on the Raspberry Pi**

Select Raspbian.

Click Install.

When the warning window pops up. Click Yes to confirm. This is just letting you know that the microSD card will be overwritten with an uncompressed version of the Raspbian operating system.

Wait for the installation process to complete.

Once the installation process is finished, Raspbian will automatically begin to boot.

**Configure your Raspberry Pi**

Click Menu in the upper left corner of the screen.

Select Preferences in the dropdown menu.

Select Raspberry Pi Configuration under Preferences.

When the configuration window appears, click on the Localisation tab.

Click on Set Locale… to set your location.

Click on Set timezone… to set your local time.

Click on Set Keyboard… to set your keyboard language.

Reconfiguring your Raspberry Pi will require a reboot. When the reboot window appears, click Yes to continue.

You are set up and ready to start using Raspberry Pi!

(https://www.imore.com/how-get-started-using-raspberry-pi)

**How to edit and compile a C program on Raspberry Pi**

To demonstrate how to create a C program, compile it, and run it on the Raspberry Pi, we’ll make a simple program that will print “hello world” in the terminal.

The coding process in C consists of four steps:

1: Creating the Source File

To start, open the Nano text editor and create a new file with a “.c” extension by entering this at the command prompt:

**sudo nano hello-world.c**

This file is where you’ll write the C code. You can write the code in any text editor, just make sure to give the file a “.c” extension.

Now, enter this code into Nano:

**#include <stdio.h>**

**int main()**

**{**

**printf("Hello, World! \n");**

**return 0;**

**}**

After entering the code, enter Ctrl-X and Y to save and exit Nano.

2: Compiling the Program

Code written in C will need to be compiled before it can be run on a computer. Compiling is the process of converting the code you write into machine readable instructions that can be understood by the computer’s processor.

When you compile your source file, a new compiled file gets created. For example, entering the command below will compile hello-world.c into a new file called myfirstcprogram:

**gcc hello-world.c -o myfirstcprogram**

3: Making the Program Executable

Now we need to make the compiled file executable. To do that, we just need to change the file permissions. Enter this at the command prompt:

**chmod +x myfirstcprogram**

4: Executing the Program

Now all we need to do to run the compiled, executable, C program is enter this at the command prompt:

**./myfirstcprogram**

(https://www.circuitbasics.com/how-to-write-and-run-a-c-program-on-the-raspberry-pi/)

Introduction to C Programming

Some advantages of learning C programming:

Easy to learn

Structured Language

It produces efficient programs

It can handle low-level activities

It can be compiled on a variety of computer platforms.

Facts about C:

C was invented to write an operating system called UNIX.

C is the successor of the language B, which was created in the early 1970’s.

C was formalized in 1988 by the American National Standard Institute (ANSI).

Today, C is the most widely used and popular System Programming Language.

Most of the state-of-the-art software have been implemented using C.

C was initially used for system development work, particularly the programs that make-up the operating system. C was adopted as a system development language because it produces code that runs nearly as fast as the code written in assembly language. Some examples of the use of C are Operating systems, language compilers, assemblers, text editors, print spoolers, network drivers, modern programs, databases, and language interpreters.

Setting up your environment:

To begin cording in C, you will need two things:

A text editor.

This is what you will use to type your program. A few examples are Windows Notepad, OS Edit command, Brief, Epsiolon, EMACS, and vim or vi. The files you create with the text editor are called your source files and they contain the program you wrote. The source files for C programs are usually named with the extension “.c”.

The C Compiler.

Simply put, the source code you write is meant to be read by humans. A compiler is then used to translate what you wrote into machine language so the CPU can execute the program. A compiler compiles source code into final executable programs. The most frequently used compilers are the SNU C/C++ compiler.

Data Types

Data types in C refer to a system of declaring variables or functions of different types. The type of variable is what determines how much space is allotted in storage and how the bit pattern stored is interpreted.

There are 4 types of types, and they are as follows:

Basic Types, which include arithmetic types and are further classified into integer types and floating-point types.

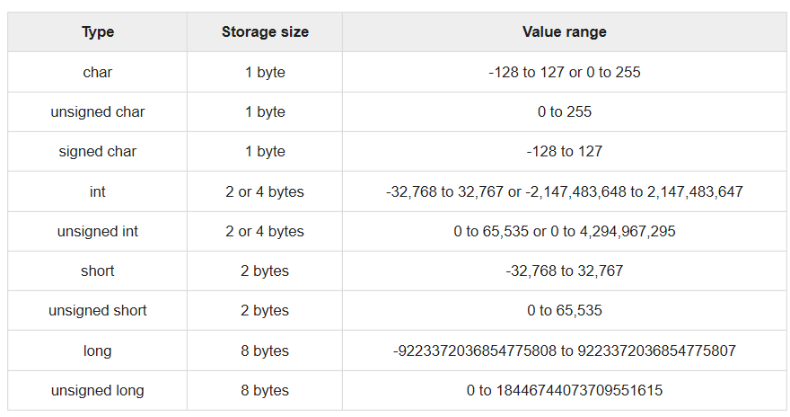
Enumerated types, which are also arithmetic types and are used to define variables that can only assign certain discrete integer values throughout the program.

The Type Void, which simply indicates that there is no value.

Derived Types, which include pointer types, array types, structure types, union types, and function types.

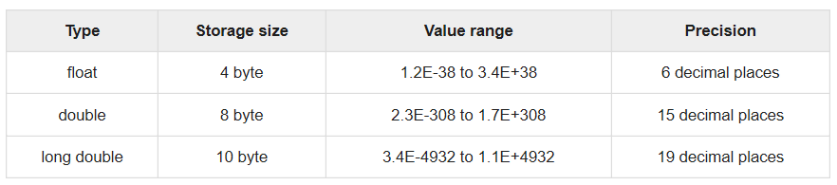
Integer Types

The following table provides the details of standard integer types with their storage sizes and range of values:



Floating-Point Types

The following table provide the details of standard floating-point types with storage sizes and value ranges and their precision:



Variables

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive. Based on the basic types explained in the previous chapter, there will be the following basic variable types:

Char: Typically a single byte. This is an integer type.

Int: The most natural size of integer for the machine.

Float: A single-precision floating point value.

Double: A double-precision floating point value.

Void: Represents the absence of a type.

Variable Definition in C

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows −

**type variable\_list;**

Here, type must be a valid C data type including char, w\_char, int, float, double, bool, or any user-defined object; and variable\_list may consist of one or more identifier names separated by commas. Some valid declarations are shown here −

**int i, j, k;**

**char c, ch;**

**float f, salary;**

**double d;**

The line int i, j, k; declares and defines the variables i, j, and k; which instruct the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows −

**type variable\_name = value;**

Some examples are −

**extern int d = 3, f = 5; // declaration of d and f.**

**int d = 3, f = 5; // definition and initializing d and f.**

**byte z = 22; // definition and initializes z.**

**char x = 'x'; // the variable x has the value 'x'.**

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables are undefined.

Variable Declaration in C

A variable declaration provides assurance to the compiler that there exists a variable with the given type and name so that the compiler can proceed for further compilation without requiring the complete detail about the variable. A variable definition has its meaning at the time of compilation only, the compiler needs actual variable definition at the time of linking the program.

A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of the program. You will use the keyword extern to declare a variable at any place. Though you can declare a variable multiple times in your C program, it can be defined only once in a file, a function, or a block of code.

Consider the following example program:

**#include <stdio.h>**

**// Variable declaration:**

**extern int a, b;**

**extern int c;**

**extern float f;**

**int main () {**

**/\* variable definition: \*/**

**int a, b;**

**int c;**

**float f;**

**/\* actual initialization \*/**

**a = 10;**

**b = 20;**

**c = a + b;**

**printf("value of c : %d \n", c);**

**f = 70.0/3.0;**

**printf("value of f : %f \n", f);**

**return 0;**

**}**

This program will produce the following result:

**value of c : 30**

**value of f : 23.333334**

The same concept applies on function declaration where you provide a function name at the time of its declaration and its actual definition can be given anywhere else. For example −

**// function declaration**

**int func();**

**int main() {**

**// function call**

**int i = func();**

**}**

**// function definition**

**int func() {**

**return 0;**

**}**

Constants

Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals.

Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal. There are enumeration constants as well.

Constants are treated just like regular variables except that their values cannot be modified after their definition.

Character Constants

Character literals are enclosed in single quotes, e.g., 'x' can be stored in a simple variable of char type.

A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in C that represent special meaning when preceded by a backslash for example, newline (\n) or tab (\t).

Here is a list of such Escape sequences:

\\ : \ character

\' : ' character

\" : " character

\? : ? character

\a : Alert or bell

\b : Backspace

\f : Form feed

\n : Newline

\r : Carriage return

\t : Horizontal tab

\v : Vertical tab

\ooo : Octal number of one to three digits

\xhh . . . : Hexadecimal number of one or more digits

Defining Constants

There are two simple ways in C to define constants −

The #define Preprocessor:

Given below is the form to use #define preprocessor to define a constant −

**#define identifier value**

The following example explains it in detail −

**#include <stdio.h>**

**#define LENGTH 10**

**#define WIDTH 5**

**#define NEWLINE '\n'**

**int main() {**

**int area;**

**area = LENGTH \* WIDTH;**

**printf("value of area : %d", area);**

**printf("%c", NEWLINE);**

**return 0;**

**}**

When the above code is compiled and executed, it produces the following result −

**value of area : 50**

The const Keyword:

You can use const prefix to declare constants with a specific type as follows −

**const type variable = value;**

The following example explains it in detail −

**#include <stdio.h>**

**int main() {**

**const int LENGTH = 10;**

**const int WIDTH = 5;**

**const char NEWLINE = '\n';**

**int area;**

**area = LENGTH \* WIDTH;**

**printf("value of area : %d", area);**

**printf("%c", NEWLINE);**

**return 0;**

**}**

When the above code is compiled and executed, it produces the following result −

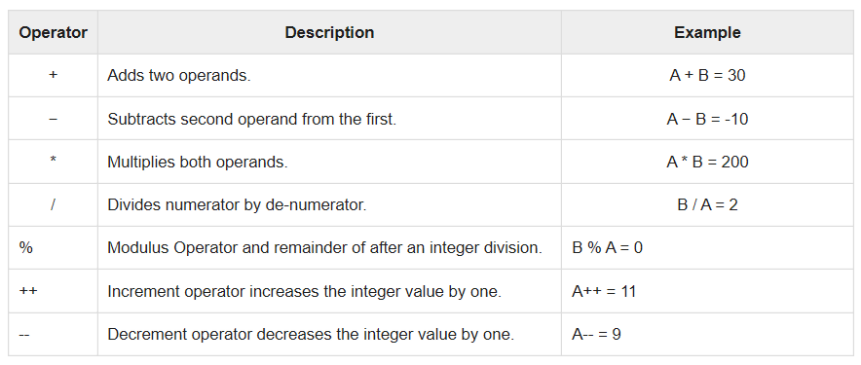
**value of area : 50**

Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators −

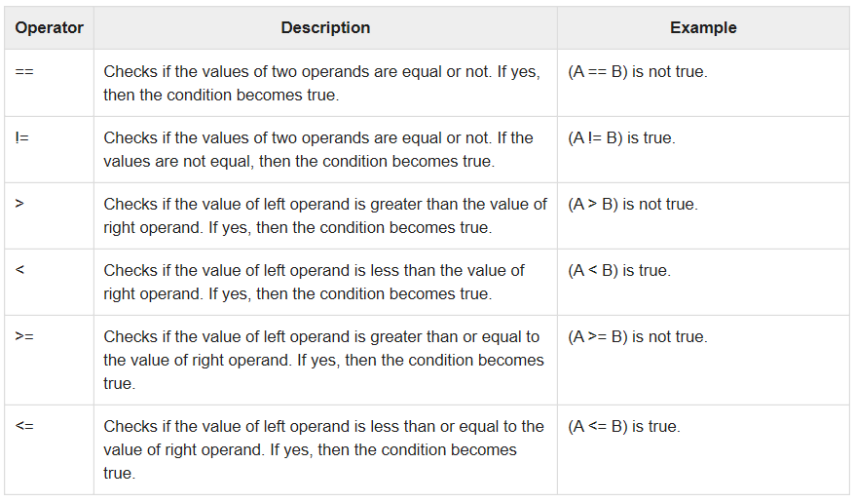
Arithmetic Operators

The following table shows all the arithmetic operators supported by the C language. Assume variable A holds 10 and variable B holds 20 then –



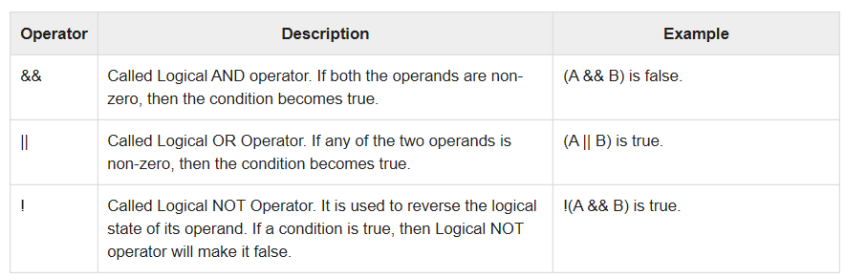
Relational Operators

The following table shows all the relational operators supported by C. Assume variable A holds 10 and variable B holds 20 then −



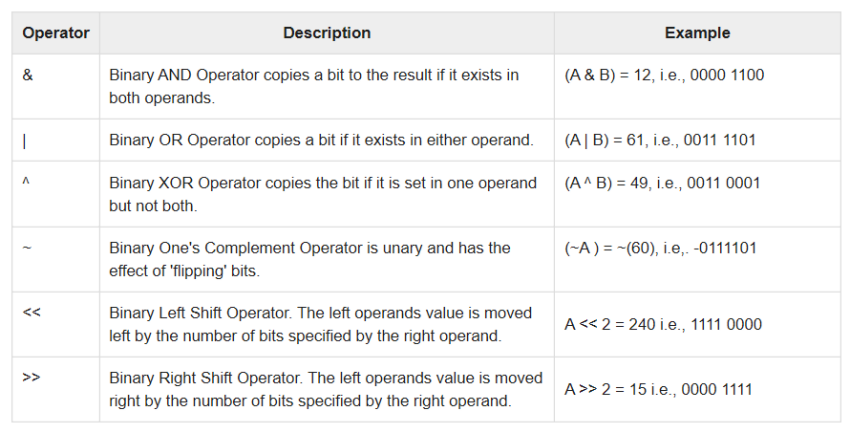
Logical Operators

Following table shows all the logical operators supported by C language. Assume variable A holds 1 and variable B holds 0, then –



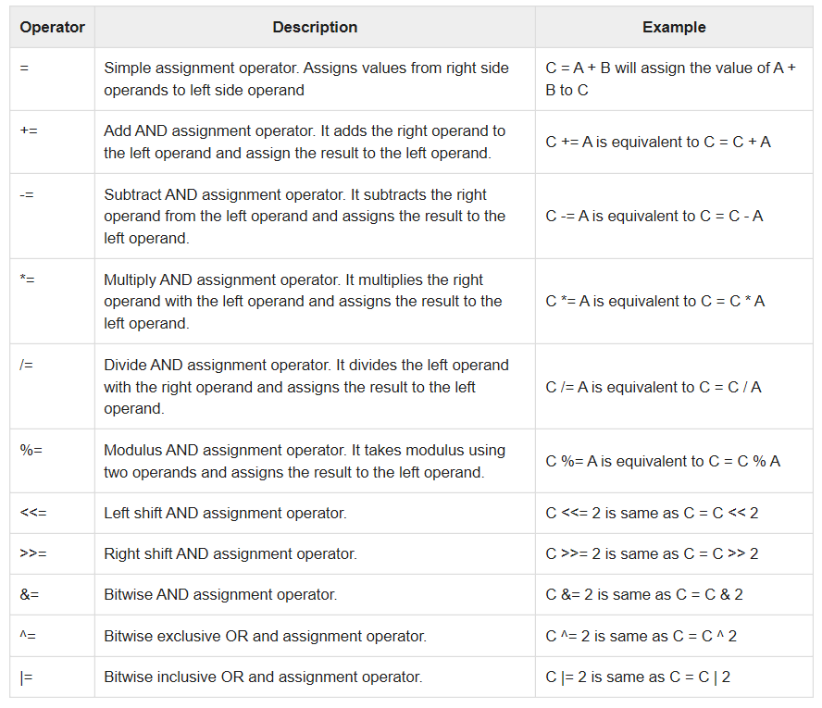
Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then −



Assignment Operators

The following table lists the assignment operators supported by the C language –

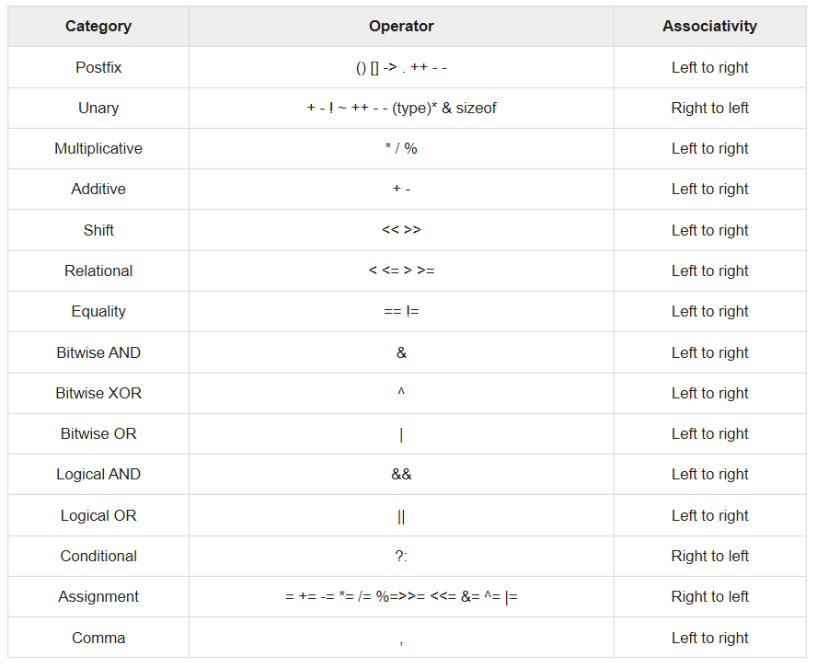


Operators Precedence in C

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has a higher precedence than the addition operator.

For example, x = 7 + 3 \* 2; here, x is assigned 13, not 20 because operator \* has a higher precedence than +, so it first gets multiplied with 3\*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.



(https://www.tutorialspoint.com/cprogramming/index.html)

Introduction to C Programming Part 2

Decision making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Show below is the general form of a typical decision-making structure found in most of the programming languages −



C programming language assumes any non-zero and non-null values as true, and if it is either zero or null, then it is assumed as false value.

C programming language provides the following types of decision making statements.

if statement :

An if statement consists of a Boolean expression followed by one or more statements.

if...else statement :

An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.

nested if statements :

You can use one if or else if statement inside another if or else if statement(s).

switch statement :

A switch statement allows a variable to be tested for equality against a list of values.

nested switch statements :

You can use one switch statement inside another switch statement(s).

The ? : Operator

The ? : Operator can be used to replace if/else statements, for example:

Exp1 ? Exp2 : Exp3;

Where Exp1, Exp2, and Exp3 are expressions.

The value of a ? expression is determined like this −

Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression.

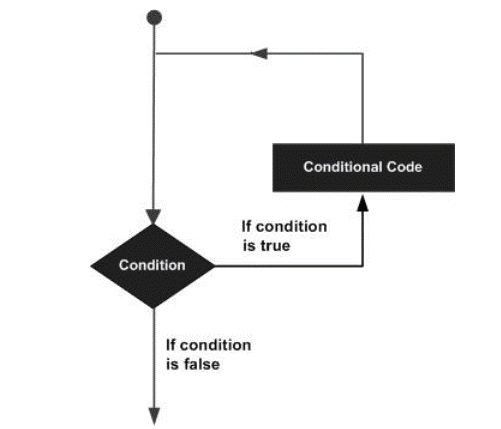
If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

Loops

You may encounter situations, when a block of code needs to be executed any number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages −



C programming language provides the following types of loops to handle looping requirements.

while loop

Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

for loop

Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

do...while loop

It is more like a while statement, except that it tests the condition at the end of the loop body.

nested loops

You can use one or more loops inside any other while, for, or do…while loop.

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C supports the following control statements:

break statement

Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.

continue statement

Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

goto statement

Transfers control to the labeled statement.

Finally, a loop becomes an infinite loop if a condition never becomes false. The for loop is traditionally used for this purpose. Since none of the three expressions that form the 'for' loop are required, you can make an endless loop by leaving the conditional expression empty.

**#include <stdio.h>**

**int main () {**

**for( ; ; ) {**

**printf("This loop will run forever.\n");**

**}**

**return 0;**

**}**

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the for(;;) construct to signify an infinite loop.

Pointers

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined −

**#include <stdio.h>**

**int main () {**

**int var1;**

**char var2[10];**

**printf("Address of var1 variable: %x\n", &var1 );**

**printf("Address of var2 variable: %x\n", &var2 );**

**return 0;**

**}**

When the above code is compiled and executed, it produces the following result −

**Address of var1 variable: bff5a400**

**Address of var2 variable: bff5a3f6**

What are Pointers?

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is −

**type \*var-name;**

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk \* used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations -

**int \*ip; /\* pointer to an integer \*/**

**double \*dp; /\* pointer to a double \*/**

**float \*fp; /\* pointer to a float \*/**

**char \*ch /\* pointer to a character \*/**

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to Use Pointers?

There are a few important operations, which we will do with the help of pointers very frequently.

(a) We define a pointer variable,

(b) assign the address of a variable to a pointer and

(c) finally access the value at the address available in the pointer variable. This is done by using unary operator \* that returns the value of the variable located at the address specified by its operand.

The following example makes use of these operations –

**#include <stdio.h>**

**int main () {**

**int var = 20; /\* actual variable declaration \*/**

**int \*ip; /\* pointer variable declaration \*/**

**ip = &var; /\* store address of var in pointer variable\*/**

**printf("Address of var variable: %x\n", &var );**

**/\* address stored in pointer variable \*/**

**printf("Address stored in ip variable: %x\n", ip );**

**/\* access the value using the pointer \*/**

**printf("Value of \*ip variable: %d\n", \*ip );**

**return 0;**

**}**

When the above code is compiled and executed, it produces the following result −

**Address of var variable: bffd8b3c**

**Address stored in ip variable: bffd8b3c**

**Value of \*ip variable: 20**

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program −

**#include <stdio.h>**

**int main () {**

**int \*ptr = NULL;**

**printf("The value of ptr is : %x\n", ptr );**

**return 0;**

**}**

When the above code is compiled and executed, it produces the following result −

**The value of ptr is 0**

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows −

**if(ptr) /\* succeeds if p is not null \*/**

**if(!ptr) /\* succeeds if p is null \*/**

(https://www.tutorialspoint.com/cprogramming/index.html)

Introduction to C Programming Part 3

Functions

A function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, strcat() to concatenate two strings, memcpy() to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

Defining a Function

The general form of a function definition in C programming language is as follows −

**return\_type function\_name( parameter list ) {**

**body of the function**

**}**

A function definition in C programming consists of a function header and a function body. Here are all the parts of a function −

Return Type − A function may return a value. The return\_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return\_type is the keyword void.

Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

Parameters − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body − The function body contains a collection of statements that define what the function does.

Given below is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum value between the two −

**/\* function returning the max between two numbers \*/**

**int max(int num1, int num2) {**

**/\* local variable declaration \*/**

**int result;**

**if (num1 > num2)**

**result = num1;**

**else**

**result = num2;**

**return result;**

**}**

Function Declarations

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

**return\_type function\_name( parameter list );**

For the above defined function max(), the function declaration is as follows −

**int max(int num1, int num2);**

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration −

**int max(int, int);**

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. For example −

**#include <stdio.h>**

**/\* function declaration \*/**

**int max(int num1, int num2);**

**int main () {**

**/\* local variable definition \*/**

**int a = 100;**

**int b = 200;**

**int ret;**

**/\* calling a function to get max value \*/**

**ret = max(a, b);**

**printf( "Max value is : %d\n", ret );**

**return 0;**

**}**

**/\* function returning the max between two numbers \*/**

**int max(int num1, int num2) {**

**/\* local variable declaration \*/**

**int result;**

**if (num1 > num2)**

**result = num1;**

**else**

**result = num2;**

**return result;**

**}**

We have kept max() along with main() and compiled the source code. While running the final executable, it would produce the following result −

**Max value is : 200**

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a function −

Call by value

This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

Call by reference

This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C uses call by value to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

Arrays

An array is a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

**type arrayName [ arraySize ];**

This is called a single-dimensional array. The arraySize must be an integer constant greater than zero and type can be any valid C data type. For example, to declare a 10-element array called balance of type double, use this statement −

**double balance[10];**

Here balance is a variable array which can hold up to 10 double numbers.

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows −

**double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};**

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

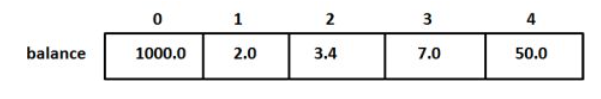
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

**double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};**

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array −

**balance[4] = 50.0;**

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above –



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

**double salary = balance[9];**

The above statement will take the 10th element from the array and assign the value to salary variable. The following example shows how to use all the three above mentioned concepts… declaration, assignment, and accessing arrays −

**#include <stdio.h>**

**int main () {**

**int n[ 10 ]; /\* n is an array of 10 integers \*/**

**int i,j;**

**/\* initialize elements of array n to 0 \*/**

**for ( i = 0; i < 10; i++ ) {**

**n[ i ] = i + 100; /\* set element at location i to i + 100 \*/**

**}**

**/\* output each array element's value \*/**

**for (j = 0; j < 10; j++ ) {**

**printf("Element[%d] = %d\n", j, n[j] );**

**}**

**return 0;**

**}**

When the above code is compiled and executed, it produces the following result −

**Element[0] = 100**

**Element[1] = 101**

**Element[2] = 102**

**Element[3] = 103**

**Element[4] = 104**

**Element[5] = 105**

**Element[6] = 106**

**Element[7] = 107**

**Element[8] = 108**

**Element[9] = 109**

Arrays in Detail

Arrays are important to C and should need a lot more attention. The following important concepts related to array should be clear to a C programmer −

Multi-dimensional arrays :

C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.

Passing arrays to functions :

You can pass to the function a pointer to an array by specifying the array's name without an index.

Return array from a function :

C allows a function to return an array.

Pointer to an array :

You can generate a pointer to the first element of an array by simply specifying the array name, without any index.

Strings

Strings are actually a one-dimensional array of characters terminated by a null character '\0'. Thus, a null-terminated string contains the characters that comprise the string followed by a null.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

**char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};**

If you follow the rule of array initialization then you can write the above statement as follows −

**char greeting[] = "Hello";**

Following is the memory presentation of the above defined string in C/C++ :



You do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string −

**#include <stdio.h>**

**int main () {**

**char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};**

**printf("Greeting message: %s\n", greeting );**

**return 0;**

**}**

When the above code is compiled and executed, it produces the following result −

**Greeting message: Hello**

C supports a wide range of functions that manipulate null-terminated strings −

strcpy(s1, s2);

Copies string s2 into string s1.

strcat(s1, s2);

Concatenates string s2 onto the end of string s1.

strlen(s1);

Returns the length of string s1.

strcmp(s1, s2);

Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.

strchr(s1, ch);

Returns a pointer to the first occurrence of character ch in string s1.

strstr(s1, s2);

Returns a pointer to the first occurrence of string s2 in string s1.

The following example uses some of the above-mentioned functions −

**#include <stdio.h>**

**#include <string.h>**

**int main () {**

**char str1[12] = "Hello";**

**char str2[12] = "World";**

**char str3[12];**

**int len ;**

**/\* copy str1 into str3 \*/**

**strcpy(str3, str1);**

**printf("strcpy( str3, str1) : %s\n", str3 );**

**/\* concatenates str1 and str2 \*/**

**strcat( str1, str2);**

**printf("strcat( str1, str2): %s\n", str1 );**

**/\* total lenghth of str1 after concatenation \*/**

**len = strlen(str1);**

**printf("strlen(str1) : %d\n", len );**

**return 0;**

**}**

When the above code is compiled and executed, it produces the following result −

**strcpy( str3, str1) : Hello**

**strcat( str1, str2): HelloWorld**

**strlen(str1) : 10**

(<https://www.tutorialspoint.com/cprogramming/index.html>)

OpenMP - Introduction

OpenMP (Open Multi-Processing) is an Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism. OpenMP is comprised of three primary API components, compiler directives, runtime library routines, and environment variables.

In the early 90's, vendors of shared-memory machines supplied similar, directive-based, Fortran programming extensions:

The user would augment a serial Fortran program with directives specifying which loops were to be parallelized

The compiler would be responsible for automatically parallelizing such loops across the SMP processors

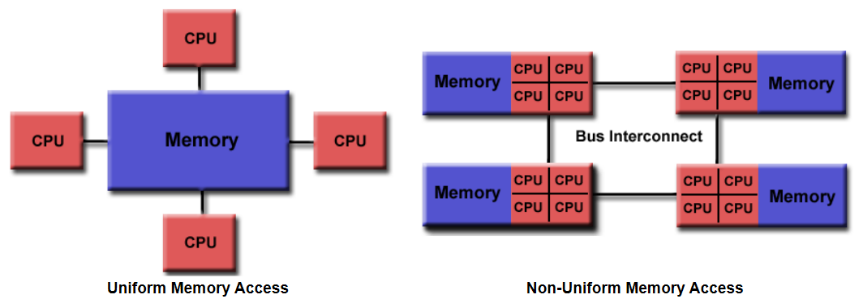
Implementations were all functionally similar, but were diverging (as usual)

First attempt at a standard was the draft for ANSI X3H5 in 1994. It was never adopted, largely due to waning interest as distributed memory machines became popular.

However, not long after this, newer shared memory machine architectures started to become prevalent, and interest resumed.

The OpenMP standard specification started in the spring of 1997, taking over where ANSI X3H5 had left off.

OpenMP is designed for multi-processor/core, shared memory machines. The underlying architecture can be shared memory UMA or NUMA.



Because OpenMP is designed for shared memory parallel programming, it largely limited to single node parallelism. Typically, the number of processing elements (cores) on a node determine how much parallelism can be implemented.

Thread-Based Parallelism:

OpenMP programs accomplish parallelism exclusively through the use of threads.

A thread of execution is the smallest unit of processing that can be scheduled by an operating system. The idea of a subroutine that can be scheduled to run autonomously might help explain what a thread is.

Threads exist within the resources of a single process. Without the process, they cease to exist.

Typically, the number of threads match the number of machine processors/cores. However, the actual use of threads is up to the application.

Explicit Parallelism:

OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.

Parallelization can be as simple as taking a serial program and inserting compiler directives....

Or as complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks.

Fork – Join Model:

OpenMP uses the fork-join model of parallel execution:



All OpenMP programs begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.

FORK: the master thread then creates a team of parallel threads.

The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.

JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.

The number of parallel regions and the threads that comprise them are arbitrary.

Data Scoping:

Because OpenMP is a shared memory programming model, most data within a parallel region is shared by default.

All threads in a parallel region can access this shared data simultaneously.

OpenMP provides a way for the programmer to explicitly specify how data is "scoped" if the default shared scoping is not desired.

As mentioned previously, OpenMP is comprised of three primary API components, compiler directives, runtime library routines, and environment variables. The application developer decides how to employ these components. In the simplest case, only a few of them are needed. Implementations differ in their support of all API components. For example, an implementation may state that it supports nested parallelism, but the API makes it clear that may be limited to a single thread - the master thread. Not exactly what the developer might expect?

OpenMP – Directives

Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise - usually by specifying the appropriate compiler flag, as discussed in the Compiling section later.

OpenMP compiler directives are used for various purposes:

Spawning a parallel region

Dividing blocks of code among threads

Distributing loop iterations between threads

Serializing sections of code

Synchronization of work among threads

Compiler directives have the following syntax:

**sentinel directive-name [clause, ...]**

For example (C/C++):

**#pragma omp parallel default(shared) private(beta,pi)**

The parallel directive

This directive defines a parallel region, which is code that will be executed by multiple threads in parallel. The following is an example of the parallel directive in C++:

**// omp\_parallel.cpp**

**// compile with: /openmp**

**#include <stdio.h>**

**#include <omp.h>**

**int main() {**

**#pragma omp parallel num\_threads(4)**

**{**

**int i = omp\_get\_thread\_num();**

**printf\_s("Hello from thread %d\n", i);**

**}**

**}**

This is the output from the code above:

**Hello from thread 0**

**Hello from thread 1**

**Hello from thread 2**

**Hello from thread 3**

For Directive

This directive causes the work done in a for loop inside a parallel region to be divided among threads. This is what the directive might look like in C++:

**#pragma omp [parallel] for [clauses]**

**for\_statement**

If parallel is also specified, clauses can be any clause accepted by the parallel or for directives, except nowait.

Sections Directive

This directive identifies code sections to be divided among all threads. For example, this directive in C++ might look like this:

**// omp\_sections.cpp**

**// compile with: /openmp**

**#include <stdio.h>**

**#include <omp.h>**

**int main() {**

**#pragma omp parallel sections num\_threads(4)**

**{**

**printf\_s("Hello from thread %d\n", omp\_get\_thread\_num());**

**#pragma omp section**

**printf\_s("Hello from thread %d\n", omp\_get\_thread\_num());**

**}**

**}**

The output for the code above would be the following:

**Hello from thread 0**

**Hello from thread 0**

Parallel For Directive

This directive is a combination of the parallel directive and the for directive. Below is example code, demonstrating this directive:

**#include <omp.h>**

**#define N 1000**

**#define CHUNKSIZE 100**

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

**int i, chunk;**

**float a[N], b[N], c[N];**

**/\* Some initializations \*/**

**for (i=0; i < N; i++)**

**a[i] = b[i] = i \* 1.0;**

**chunk = CHUNKSIZE;**

**#pragma omp parallel for \**

**shared(a,b,c,chunk) private(i) \**

**schedule(static,chunk)**

**for (i=0; i < N; i++)**

**c[i] = a[i] + b[i];**

**}**

Parallel Sections Directive

The parallel sections directive provides a shortcut form for specifying a parallel region that has only a single sections directive. The semantics are the same as explicitly specifying a parallel directive immediately followed by a sections directive. The syntax of the parallel sections directive is as follows:

**#pragma omp parallel sections [clause[[,] clause] ...] new-line**

**{**

**[#pragma omp section new-line]**

**structured-block**

**[#pragma omp section new-linestructured-block ]**

**...**

**}**

Environment Variables:

OpenMP provides several environment variables for controlling the execution of parallel code at run-time. These environment variables can be used to control such things as:

Setting the number of threads

Specifying how loop iterations are divided

Binding threads to processors

Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism

Enabling/disabling dynamic threads

Setting thread stack size

Setting thread wait policy

Setting OpenMP environment variables is done the same way you set any other environment variables and depends upon which shell you use. For example:

csh/tcsh

**setenv OMP\_NUM\_THREADS 8**

sh/bash

**export OMP\_NUM\_THREADS=8**

Below is an example of OpenMP code structure in (C/C++):

**#include <omp.h>**

**main () {**

**int var1, var2, var3;**

**Serial code**

**.**

**.**

**.**

**Beginning of parallel region. Fork a team of threads.**

**Specify variable scoping**

**#pragma omp parallel private(var1, var2) shared(var3)**

**{**

**Parallel region executed by all threads**

**.**

**Other OpenMP directives**

**.**

**Run-time Library calls**

**.**

**All threads join master thread and disband**

**}**

**Resume serial code**

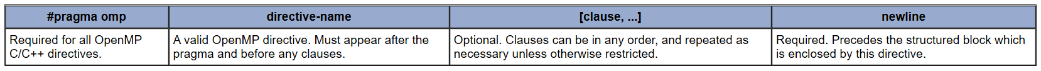
**.**

**.**

**.**

**}**

C/C++ Directives Format



Example: #pragma omp parallel default(shared) private(beta,pi)

These are the general rules:

Case sensitive

Directives follow conventions of the C/C++ standards for compiler directives

Only one directive-name may be specified per directive

Each directive applies to at most one succeeding statement, which must be a structured block.

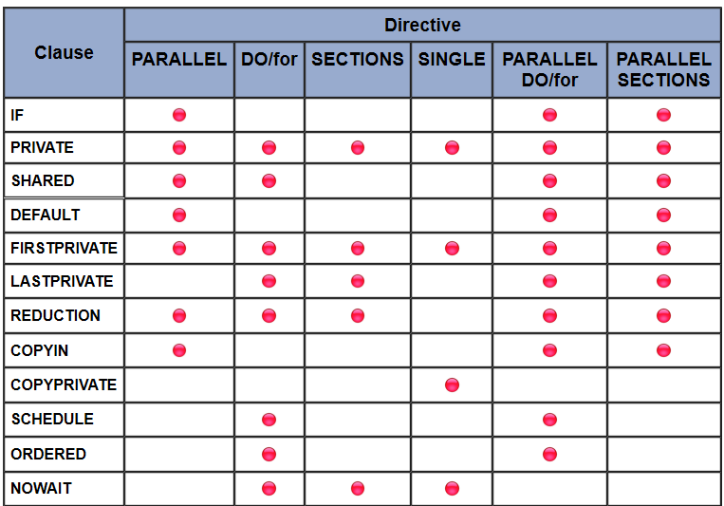
Long directive lines can be "continued" on succeeding lines by escaping the newline character with a backslash ("\") at the end of a directive line.

(https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-directives?view=vs-2019#sections-openmp)

OpenMP - Clauses

Directive Clauses

The table below summarizes which clauses are accepted by which OpenMP directives:



The following OpenMP directives do not accept clauses:

MASTER

CRITICAL

BARRIER

ATOMIC

FLUSH

ORDERED

THREADPRIVATE

Data Scope Attribute Clauses are used in conjunction with several directives (PARALLEL, DO/for, and SECTIONS) to control the scoping of enclosed variables.

These constructs provide the ability to control the data environment during execution of parallel constructs.

They define how and which data variables in the serial section of the program are transferred to the parallel regions of the program (and back)

They define which variables will be visible to all threads in the parallel regions and which variables will be privately allocated to all threads.

for and its clause: private, shared, reduction

The IF clause simply specifies whether a loop should be executed in parallel or in serial.

The PRIVATE clause declares variables in its list to be private to each thread.

PRIVATE variables behave as follows:

A new object of the same type is declared once for each thread in the team

All references to the original object are replaced with references to the new object

Should be assumed to be uninitialized for each thread

The SHARED clause declares variables in its list to be shared among all threads in the team.

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)

The DEFAULT clause allows the user to specify a default scope for all variables in the lexical extent of any parallel region.

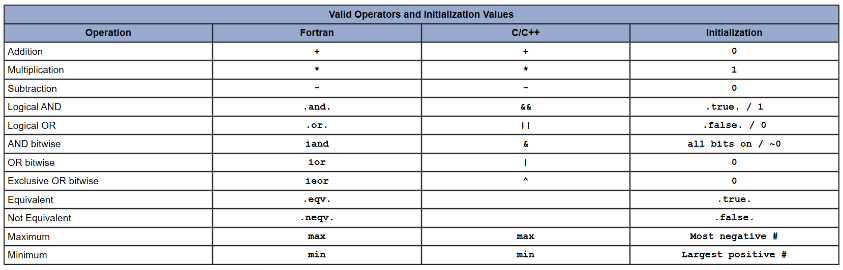
Specific variables can be exempted from the default using the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses

The C/C++ OpenMP specification does not include private or firstprivate as a possible default. However, actual implementations may provide this option.

Using NONE as a default requires that the programmer explicitly scope all variables.

The REDUCTION clause performs a reduction operation on the variables that appear in its list.

A private copy for each list variable is created and initialized for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.



A parallel region has at least one barrier, at its end, and may have additional barriers within it. At each barrier, the other members of the team must wait for the last thread to arrive. To minimize this wait time, shared work should be distributed so that all threads arrive at the barrier at about the same time. If some of that shared work is contained in for constructs, the SCHEDULE clause can be used for this purpose.

The code below demonstrates the SCHEDULE clause:

**#pragma omp parallel for schedule(dynamic)**

**for(i=0; i<n; i++) {**

**unpredictable\_amount\_of\_work(i);**

**}**

The ORDERED clause is required on a parallel for statement if an ordered directive is to be used in the loop. The following code demonstrates the ORDERED clause:

**#pragma omp parallel for private(myval) ordered**

**{**

**for(i=1; i<=n; i++){**

**myval = do\_lots\_of\_work(i);**

**#pragma omp ordered**

**{**

**printf("%d %d\n", i, myval);**

**}**

**}**

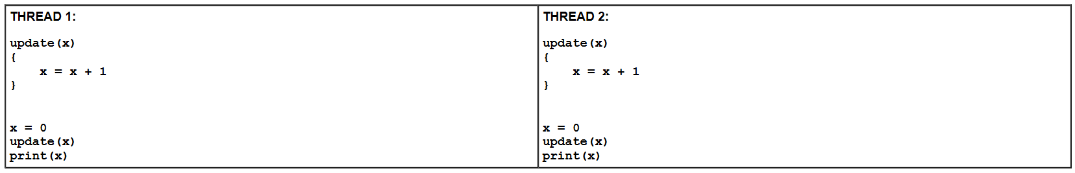
**}**

(<https://computing.llnl.gov/tutorials/openMP/#Introduction>)

OpenMP - directives and other function calls

Synchronization Constructs

Consider a simple example where two threads are both trying to update variable x at the same time:



One possible execution sequence:

Thread 1 initializes x to 0 and calls update(x)

Thread 1 adds 1 to x.

x now equals 1

Thread 2 initializes x to 0 and calls update(x)

x now equals 0

Thread 1 prints x, which is equal to 0 instead of 1

Thread 2 adds 1 to x.

x now equals 1.

Thread 2 prints x as 1.

To avoid a situation like this, the updating of x must be synchronized between the two threads to ensure that the correct result is produced.

OpenMP provides a variety of Synchronization Constructs that control how the execution of each thread proceeds relative to other team threads.

The MASTER directive specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code.

There is no implied barrier associated with this directive. This is the format for the MASTER directive in C/C++:

**#pragma omp master newline**

**structured\_block**

The CRITICAL directive specifies a region of code that must be executed by only one thread at a time. If a thread is currently executing inside a CRITICAL region and another thread reaches that CRITICAL region and attempts to execute it, it will block until the first thread exits that CRITICAL region.

This is the format for the CRITICAL directive in C/C++:

**#pragma omp critical [ name ] newline**

**structured\_block**

The BARRIER directive synchronizes all threads in the team.

When a BARRIER directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.

This is the format for the BARRIER directive in C/C++:

**#pragma omp barrier newline**

The ORDERED directive specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor.

Threads will need to wait before executing their chunk of iterations if previous iterations haven't completed yet.

Used within a DO / for loop with an ORDERED clause

The ORDERED directive provides a way to "fine tune" where ordering is to be applied within a loop. Otherwise, it is not required.

This is the format for the ORDERED directive in C/C++:

**#pragma omp for ordered [clauses...]**

**(loop region)**

**#pragma omp ordered newline**

**structured\_block**

**(endo of loop region)**

Run-Time Library Routines

The OpenMP API includes an ever-growing number of run-time library routines.

These routines are used for a variety of purposes as shown in the table below:



The following are a few important Run-Time Library Routines and a more in-depth explanation…

omp\_set\_num\_threads()

Sets the number of threads that will be used in the next parallel region. Must be a postive integer.

Format(C/C++):

**#include <omp.h>**

**void omp\_set\_num\_threads(int num\_threads)**

omp\_get\_num\_threads()

Returns the number of threads that are currently in the team executing the parallel region from which it is called.

Format(C/C++):

**#include <omp.h>**

**int omp\_get\_num\_threads(void)**

omp\_get\_max\_threads()

Returns the maximum value that can be returned by a call to the OMP\_GET\_NUM\_THREADS function.

Format(C/C++):

**#include <omp.h>**

**int omp\_get\_max\_threads(void)**

omp\_get\_thread\_num()

Returns the thread number of the thread, within the team, making this call. This number will be between 0 and OMP\_GET\_NUM\_THREADS-1. The master thread of the team is thread 0

Format(C/C++):

**#include <omp.h>**

**int omp\_get\_thread\_num(void)**

omp\_get\_num\_procs()

Returns the number of processors that are available to the program.

Format(C/C++):

**#include <omp.h>**

**int omp\_get\_num\_procs(void)**

omp\_in\_parallel()

May be called to determine if the section of code which is executing is parallel or not.

Format(C/C++):

**#include <omp.h>**

**int omp\_in\_parallel(void)**

(<https://computing.llnl.gov/tutorials/openMP/#Introduction>)

MPI

The Message Passing Interface Standard (MPI) is a message passing library standard based on the consensus of the MPI Forum, which has over 40 participating organizations, including vendors, researchers, software library developers, and users. The goal of the Message Passing Interface is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs. As such, MPI is the first standardized, vendor independent, message passing library. The advantages of developing message passing software using MPI closely match the design goals of portability, efficiency, and flexibility. MPI is not an IEEE or ISO standard, but has in fact, become the "industry standard" for writing message passing programs on HPC platforms.

MPI is a specification for the developers and users of message passing libraries. By itself, it is NOT a library - but rather the specification of what such a library should be.

MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process.

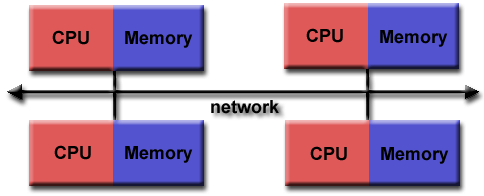
Simply stated, the goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs. The interface attempts to be practical, portable, efficient, and flexible.

The MPI standard has gone through a number of revisions, with the most recent version being MPI-3.x

Actual MPI library implementations differ in which version and features of the MPI standard they support.

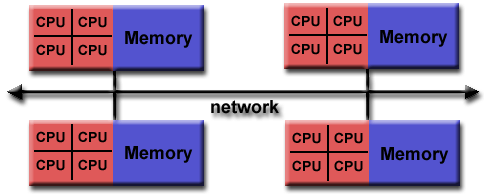
Programming Model:

Originally, MPI was designed for distributed memory architectures, which were becoming increasingly popular at that time (1980s - early 1990s).



As architecture trends changed, shared memory SMPs were combined over networks creating hybrid distributed memory / shared memory systems.

MPI implementors adapted their libraries to handle both types of underlying memory architectures seamlessly. They also adapted/developed ways of handling different interconnects and protocols.



Today, MPI runs on virtually any hardware platform:

Distributed Memory

Shared Memory

Hybrid

The programming model clearly remains a distributed memory model however, regardless of the underlying physical architecture of the machine.

All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.

Reasons for Using MPI:

Standardization - MPI is the only message passing library that can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.

Portability - There is little or no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.

Performance Opportunities - Vendor implementations should be able to exploit native hardware features to optimize performance. Any implementation is free to develop optimized algorithms.

Functionality - There are over 430 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.

Availability - A variety of implementations are available, both vendor and public domain.

MPI Functions

MPI\_Init()

This function is used to initialize the MPI execution environment. This routine must be called before any other MPI routine. It must be called at most once; subsequent calls are erroneous. For example:

**int MPI\_Init(**

**int \*argc,**

**char \*\*\*argv**

**);**

**int MPI\_Init(**

**int \*argc,**

**wchar\_t \*\*\*argv**

**);**

MPI\_Finalize()

This function terminates the MPI execution environment.

**int MPI\_Finalize( void );**

This routine cleans up all MPI state. Once this routine is called, no MPI routine (even MPI\_INIT) may be called. The user must ensure that all pending communications involving a process completes before the process calls MPI\_FINALIZE.

All processes must call this routine before exiting. The number of processes running after this routine is called is undefined; it is best not to perform much more than a return rc after calling MPI\_Finalize.

MPI\_Comm\_size()

This function determines the size of the group associated with a communicator.

**int MPI\_Comm\_size(**

**MPI\_Comm comm,**

**int \*size**

**);**

MPI\_Comm\_rank()

This function indicates the rank of the process that calls it in the range from size-1, where size is the return value of MPI\_COMM\_SIZE.

**int MPI\_Comm\_rank(**

**MPI\_Comm comm,**

**int \*rank**

**);**

MPI\_Send()

This function performs a blocking send. It may block until the message is received by the destination process.

**int MPI\_Send(**

**void \*buf,**

**int count,**

**MPI\_Datatype datatype,**

**int dest,**

**int tag,**

**MPI\_Comm comm**

**);**

Parameters:

buf

[in] initial address of send buffer (choice)

count

[in] number of elements in send buffer (nonnegative integer)

datatype

[in] datatype of each send buffer element (handle)

dest

[in] rank of destination (integer)

tag

[in] message tag (integer)

comm

[in] communicator (handle)

MPI\_Recv()

This function performs a blocking receive for a message. The receive buffer consists of the storage containing count consecutive elements of the type specified by datatype, starting at address buf. The length of the received message must be less than or equal to the length of the receive buffer. An overflow error occurs if all incoming data does not fit, without truncation, into the receive buffer.

**int MPI\_Recv(**

**void \*buf,**

**int count,**

**MPI\_Datatype datatype,**

**int source,**

**int tag,**

**MPI\_Comm comm,**

**MPI\_Status \*status**

**);**

Parameters

buf

[out] initial address of receive buffer (choice)

count

[in] maximum number of elements in receive buffer (integer)

datatype

[in] datatype of each receive buffer element (handle)

source

[in] rank of source (integer)

tag

[in] message tag (integer)

comm

[in] communicator (handle)

status

[out] status object (Status)

(<https://computing.llnl.gov/tutorials/mpi/>)

MPI II, Other Functions

MPI\_Get\_version()

This function returns the version number of MPI.

**int MPI\_Get\_version(**

**int \*version,**

**int \*subversion**

**);**

MPI\_Get\_processor\_name()

This function gets the name of the processor. The name returned should identify a particular piece of hardware; the exact format is implementation defined. This name may or may not be the same as might be returned by gethostname, uname, or sysinfo.

**int MPI\_Get\_processor\_name(**

**char \*name,**

**int \*resultlen**

**);**

**int MPI\_Get\_processor\_name(**

**wchar\_t \*name,**

**int \*resultlen**

**);**

MPI\_Sendrecv()

This function sends and receives a message.

**int MPI\_Sendrecv(**

**void \*sendbuf,**

**int sendcount,**

**MPI\_Datatype sendtype,**

**int dest,**

**int sendtag,**

**void \*recvbuf,**

**int recvcount,**

**MPI\_Datatype recvtype,**

**int source,**

**int recvtag,**

**MPI\_Comm comm,**

**MPI\_Status \*status**

**);**

Parameters

sendbuf

[in] initial address of send buffer (choice)

sendcount

[in] number of elements in send buffer (integer)

sendtype

[in] type of elements in send buffer (handle)

dest

[in] rank of destination (integer)

sendtag

[in] send tag (integer)

recvbuf

[out] initial address of receive buffer (choice)

recvcount

[in] number of elements in receive buffer (integer)

recvtype

[in] type of elements in receive buffer (handle)

source

[in] rank of source (integer)

recvtag

[in] receive tag (integer)

comm

[in] communicator (handle)

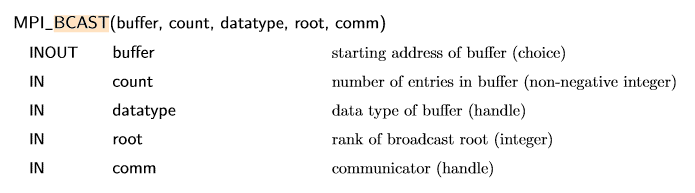
status

[out] status object (Status). This refers to the receive operation.

(<https://computing.llnl.gov/tutorials/mpi/>)

MPI III, Collective Communication Part 1

MPI\_Bcast

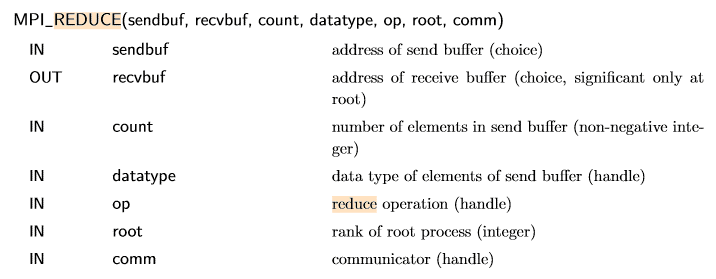


If comm is an intracommunicator, MPI\_BCAST broadcasts a message from the process with rank root to all processes of the group, itself included. It is called by all members of the group using the same arguments for comm and root. On return, the content of root’s buﬀer is copied to all other processes.

General, derived datatypes are allowed for datatype. The type signature of count, datatype on any process must be equal to the type signature of count, datatype at the root. This implies that the amount of data sent must be equal to the amount received, pairwise between each process and the root. MPI\_BCAST and all other data-movement collective routines make this restriction. Distinct type maps between sender and receiver are still allowed.

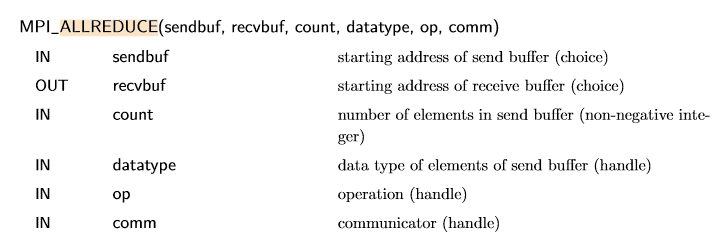
MPI\_Reduce

The reduction operation can be either one of a predeﬁned list of operations, or a user-deﬁned operation. The global reduction functions come in several ﬂavors: a reduce that returns the result of the reduction to one member of a group, an all-reduce that returns this result to all members of a group, and two scan (parallel preﬁx) operations. In addition, a reduce-scatter operation combines the functionality of a reduce and of a scatter operation.



MPI\_Allreduce

MPI includes a variant of the reduce operations where the result is returned to all processes in a group. MPI requires that all processes from the same group participating in these operations receive identical results.



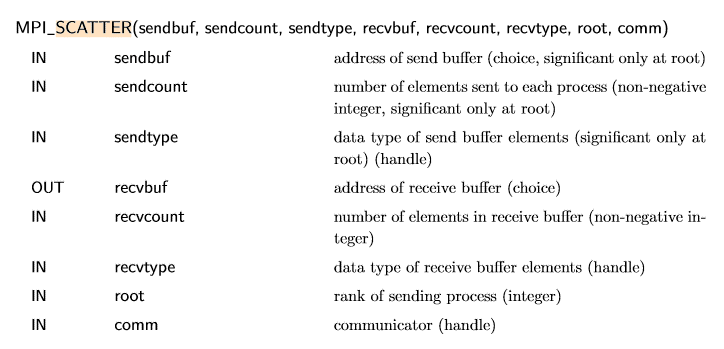
MPI\_Barrier

If comm is an intracommunicator, MPI\_BARRIER blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call. If comm is an intercommunicator, MPI\_BARRIER involves two groups. The call returns at processes in one group (group A) of the intercommunicator only after all members of the other group (group B) have entered the call (and vice versa). A process may return from the call before all processes in its own group have entered the call.



MPI\_Scatter

MPI\_SCATTER is the inverse operation to MPI\_GATHER. If comm is an intracommunicator, the outcome is as if the root executed n send operations, and each process executed a receive.

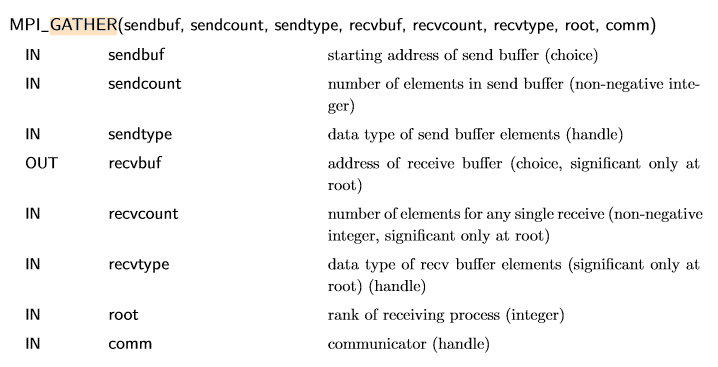


(https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf)

MPI IV, Collective Communication Part 2

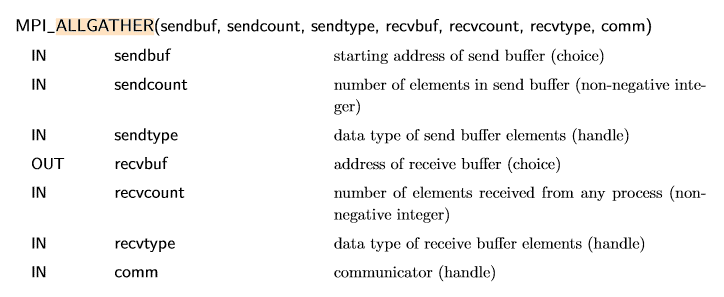
MPI\_Gather

If comm is an intracommunicator, each process (root process included) sends the contents of its send buﬀer to the root process. The root process receives the messages and stores them in rank order.



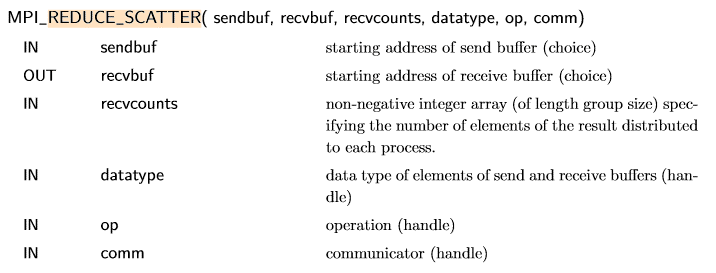
MPI\_Allgather

MPI\_ALLGATHER can be thought of as MPI\_GATHER, but where all processes receive the result, instead of just the root. The block of data sent from the j-th process is received by every process and placed in the j-th block of the buﬀer recvbuf. The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process.



MPI\_Reduce\_scatter

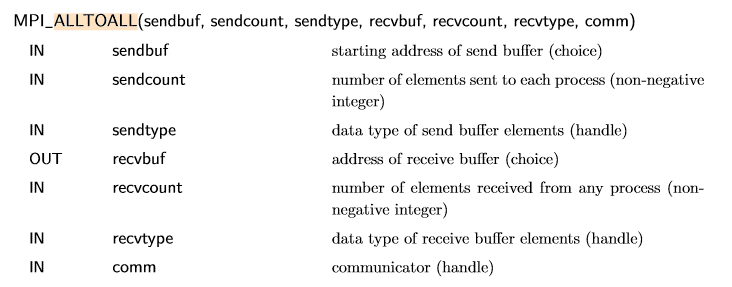
MPI\_REDUCE\_SCATTER extends the functionality of MPI\_REDUCE\_SCATTER\_BLOCK such that the scattered blocks can vary in size. Block sizes are determined by the recvcounts array, such that the i-th block contains recvcounts[i] elements.



If comm is an intracommunicator, MPI\_REDUCE\_SCATTER ﬁrst performs a global, element-wise reduction on vectors of count =Pn−1 i=0 recvcounts[i] elements in the send buﬀers deﬁned by sendbuf, count and datatype, using the operation op, where n is the number of processes in the group of comm. The routine is called by all group members using the same arguments for recvcounts, datatype, op and comm. The resulting vector is treated as n consecutive blocks where the number of elements of the i-th block is recvcounts[i]. The blocks are scattered to the processes of the group. The i-th block is sent to process i and stored in the receive buﬀer deﬁned by recvbuf, recvcounts[i] and datatype.

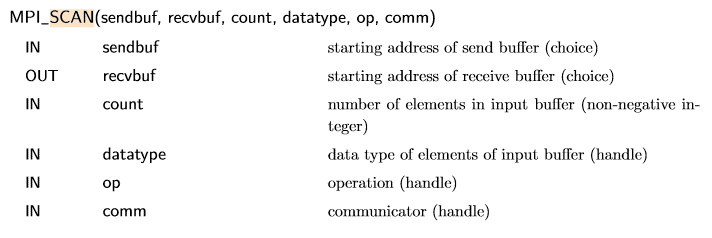
MPI\_Alltoall

MPI\_ALLTOALL is an extension of MPI\_ALLGATHER to the case where each process sends distinct data to each of the receivers. The j-th block sent from process i is received by process j and is placed in the i-th block of recvbuf. The type signature associated with sendcount, sendtype, at a process must be equal to the type signature associated with recvcount, recvtype at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. As usual, however, the type maps may be diﬀerent.



MPI\_Scan

If comm is an intracommunicator, MPI\_SCAN is used to perform a preﬁx reduction on data distributed across the group. The operation returns, in the receive buﬀer of the process with rank i, the reduction of the values in the send buﬀers of processes with ranks 0,...,i (inclusive). The routine is called by all group members using the same arguments for count, datatype, op and comm, except that for user-deﬁned operations, the same rules apply as for MPI\_REDUCE. The type of operations supported, their semantics, and the constraints on send and receive buﬀers are as for MPI\_REDUCE.



(https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf)