CPA – Parallel Computing

Degree in Computer Science

S1. Introduction to Parallel Programming Environments

J. M. Alonso, P. Alonso, F. Alvarruiz, I. Blanquer,
J. Ibáñez, E. Ramos, J. E. Román

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Year 2024/25

Contents

1 Programming in C
- A Brief Reminder of C

2 Usage of Parallel Computers
- Development Cycle

Section 1

Programming in C

- A Brief Reminder of C

C Language

C is a general purpose programming language

- Features: compiled, portable and efficient
- Java and C++ inherit the syntax of C
- A simple language kernel, additional functionality by means of software libraries
- One of the most common languages in supercomputing

```
void daxpy(int n, double a, double *x, double *y)
{
    int i;
    for (i=0; i<n; i++) {
        y[i] = a*x[i] + y[i];
    }
}
```

Variables and Basic Types

All variables used must be previously declared

- Integer: char, int, long; modifier unsigned
- Enumerates: enum (equivalent to an integer)
- Floating Point: float, double
- Void type: void (special usage)
- Derived types: struct, arrays, pointers

```
char c;
int i1,i2;
enum {NORTH,SOUTH,EAST,WEST} dir;
unsigned int k;
const float pi=3.141592;
double r=2.5,g;
```

```
c = ’M’;
i1 = 2;
i2 = -5*i1;
dir = SOUTH;
k = (unsigned int) dir;
g = 2*pi*r;
```

New types can be defined using typedef

```
typedef enum {RED,GREEN,BLUE,YELLOW,WHITE,BLACK} color;
color c1,c2;
```

Statements and Expressions

There are different types of statements:

- Type or variable declaration (inside/outside the function)
- Expression, typically an assignment var=expr
- Compound statement ({...} block)
- Conditions (if, switch), loops (for, while, do)
- Others: void statement (;), jump (goto)

Expressions:

- Assignments: =, +=, -=, *=, /=; increments: ++, --
- Arithmetic: +, -, *, /, %; bit-wise: ~, &, |, ^, <<, >>
- Logic: ==, !=, <, >, <=, >=, ||, &&, !
  Zero means “false” and any other value stands for “true”
- Ternary operator: a?b:c

Examples of Flow Control Constructs

```
if (j>0) value = 1.0;
else value = -1.0;
```

```
if (i>1 && (qi[i]-1.0)<1e-7) {
  zm1[i] *= 1.0+sk1[i-1];
  zm2[i] *= 1.0+sk1[i-1];
} else {
  zm1[i] *= 1.0+sk0[i-1];
  zm2[i] *= 1.0+sk0[i-1];
}
```

```
for (i=0;i<n;i++) x[i] = 0.0;
```

```
k = 0;
while (k<n) {
  if (a[k]<0.0) break;
  z[k] = 2.0*sqrt(a[k]);
  k++;
}
```

```
switch (dir) {
  case NORTH:
    y += 1; break;
  case SOUTH:
    y -= 1; break;
  case EAST:
    x += 1; break;
  case WEST:
    x -= 1; break;
}
```

```
for (i=0;i<n;i++) {
  y[i] = b[i];
  for (j=0;j<i;j++) {
    y[i] -= L[i][j]*y[j];
  }
  y[i] /= L[i][i];
}
```

Arrays and Pointers

Array: collection of variables of the same type

- Length defined in the declaration
- Elements accessed by an index (starting in 0)

```
#define N 10
int i;
double a[N],s=0.0;
for (i=0;i<N;i++)
  s = s + a[i];
```

Multidimensional arrays: `double matriz[N][M];`
Strings are arrays of type `char` ending with the character `'\0'`

Pointer: variable containing the address of another variable

- In the declaration, `*` is added before the name of the variable
- Operator `&` returns the address of a variable
- Operator `*` enables accessing the value pointed to

```
double a[4] =
 {1.1,2.2,3.3,4.4};
double *p,x;
p = &a[2];
x = *p;
*p = 0.0;
p = a;  /* &a[0] */
```

More about Pointers

Pointer arithmetic

- Basic operations: +, −, ++
- The step length is equal to the pointed type size

```
char s[] =
    "Parallel Computing";
char *p = s;
while (*p!=’C’) p++;
```

Null pointer

- Its value is zero (`NULL`)
- Normally used to indicate a failure

```
double w,*p;
...
if (!p)
  error("Invalid Pointer");
else w = *p;
```

Generic pointer

- Type: `void*`
- It can be casted to point to a variable of any type

```
void *p;
double x=10.0,z;
p = &x;
z = *(double*)p;
```

Multiple level pointer: `double **p` (pointer to pointer)

Structures

Structure: a collection of heterogeneous data

- Members can be accessed with `.` (or `->` in the case of struct pointers)

```
struct complex {
  double re,im;
};
struct complex c1, *c2;
c1.re = 1.0;
c1.im = 2.0;
c2 = &c1;
c2->re = -1.0;
```

```
typedef struct {
  int i,j,k;
  const char *label;
  double data[100];
} mystruct;

mystruct s;
s.label = "NEW";
```

Functions

A C program has at least one function (main)

Functions return a value (unless the function type was void)

```
double rad2deg(double x) {
  return x*57.29578;
}
```

```
void message(int k) {
 printf("End stage %d\n",k);
}
```

Arguments are passed by value (arguments by reference can be achieved via pointers)

```
float fun1(float a,float b){
  float c;
  c = (a+b)/2.0;
  return c;
}
...
w = fun1(6.0,6.5);
```

```
void fun2(float *a,float *b){
  float c;
  c = ((*a)+(*b))/2.0;
  if (fun3(c)*fun3(*a)<=0.0)
    *b = c;
  else *a = c;
}
...
fun2(&x,&y);
```

Functions can be declared before their definition (prototypes)

Library functions

String processing <string.h>

- String copy (strcpy), string compare (strcmp)
- Memory copy (memcpy), memory set (memset)

Input-Output <stdio.h>

- Standard: printf, scanf
- Files: fopen, fclose, fprintf, fscanf

Standard tools <stdlib.h>

- Dynamic memory management: malloc, free
- Conversions: atof, atoi

Mathematical functions <math.h>

- Functions and operations: sin, cos, exp, log, pow, sqrt
- Rounding: floor, ceil, fabs

Example with Files

```
#include <stdio.h>
#include <stdlib.h>

void readdata( char *filename )
{
  FILE *fd;
  int i,n,*ia,*ja;
  double *va;
  fd = fopen(filename,"r");
  if (!fd) {
    perror("Error - fopen");
    exit(1);
  }
  fscanf(fd,"%i",&n);        /* number of data to be read */
  ia = (int*) malloc(n*sizeof(int));
  ja = (int*) malloc(n*sizeof(int));
  va = (double*) malloc(n*sizeof(double));
  for (i=0;i<n;i++) {
    fscanf(fd,"%i%i%lf",ia+i,ja+i,va+i);
  }
  fclose(fd);
  process(n,ia,ja,va);
  free(ia); free(ja); free(va);
}
```

Variable Types

Global variables

- They are declared outside any function block
- They can be accessed from any point of the program
- They are allocated in the data segment

Local variables

- They are declared inside a function block
- Scope restricted to the block
- They are created in the *stack*, and destroyed at exit

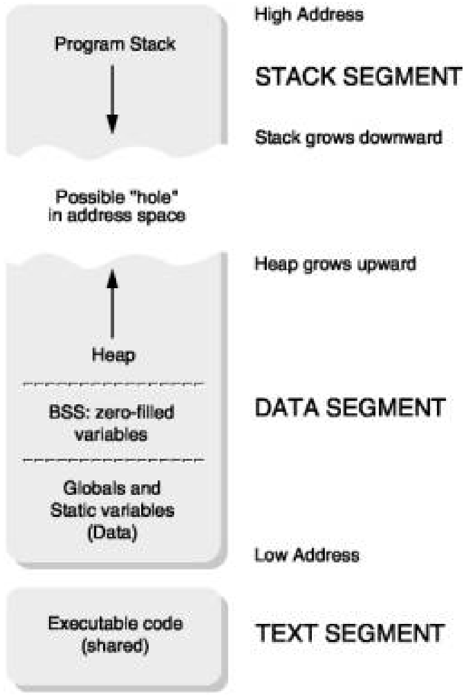
Static variables

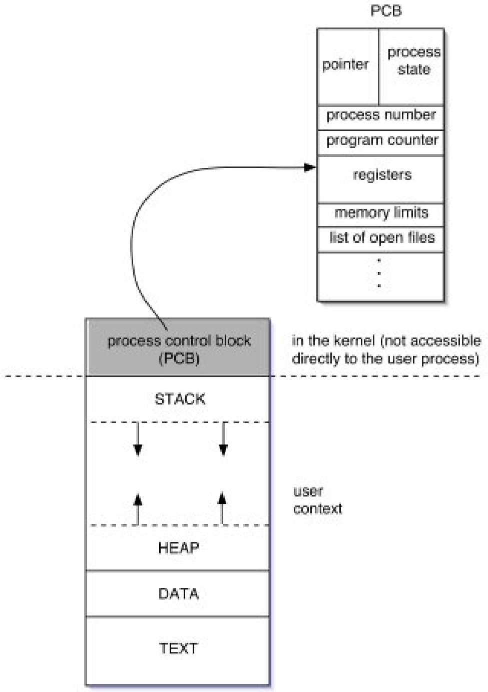
- With `static` modifier
- Local scope but persistent between successive calls

Variables allocated in dynamic memory

- Created with `malloc`, persist until `free` is called
- They are created in the *heap*

Memory Model of Unix Processes



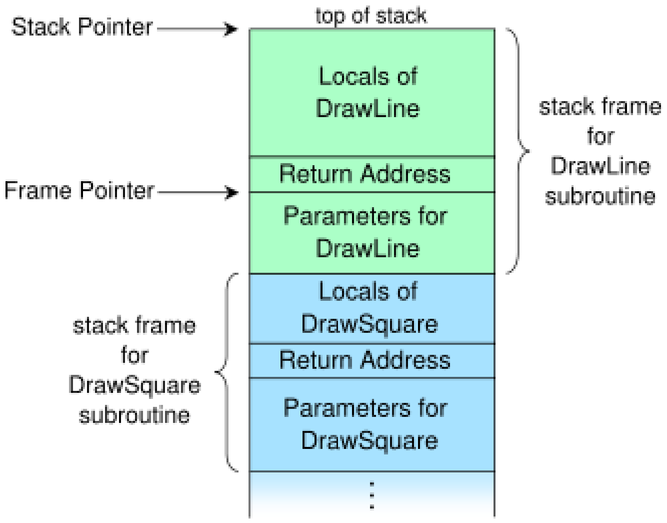


Call Stack

The arguments of a function are treated as local variables

When entering a function:

1. all the arguments are inserted in the stack
2. the return address is stacked also
3. local variables are created in the stack



When executing `return` or leaving the function, the whole context created is destroyed

Section 2

Usage of Parallel Computers

- Development Cycle

Development Cycle

The compilation process consist of:

- Preprocessing: the C source code is modified according to a set of instructions (preprocessing directives)
- Compilation: the object code (binary) is created from the already preprocessed code
- Linking: it merges the object codes from the different modules and external libraries to generate the final executable

The development cycle is completed with the following steps:

- Automation of complex program compiling (`make`)
- Error debugging (`gdb`, `valgrind`)
- Performance analysis (`gprof`)

Preprocessing

Before compiling, preprocessing takes place (cpp command, automatically invoked)

- include: inserts the contents of another file
- define: defines constants and macros (with arguments)
- if, ifdef: conditional compilation
- pragma: compiler directive

```
#include "myheader.h"

#define PI 3.141592
#define DEBUG_
#define AVG(a,b) ((a)+(b))/2

#ifdef DEBUG_
  printf("variable i=%d\n",i);
#endif
```

Compiling and Linking

Compiling: cc

- For each *.c file, a *.o object file is generated
- Contains the machine code of the functions and variables, as well as a list of unresolved symbols

Linking: ld

- Resolves all unresolved dependencies using the *.o files and external libraries (*.a, *.so)

ex.c

```
#include <stdio.h>
extern double f1(double);
int main() {
  double x = f1(4.5);
  printf("x = %g\n",x);
  return 0;
}
```

f1.c

```
#include <math.h>
double f1(double x) {
  return 2.0/(1.0+log(x));
}
```

```
$ gcc -o ex ex.c f1.c -lm
```

Compilation of Parallel Programs

OpenMP is based on directives `#pragma omp`

- A compiler without OpenMP support ignores these directives
- Recent compilers have support, usually with a special option when compiling and linking

```
$ gcc -fopenmp -o prgomp prgomp.c
```

MPI provides the `mpicc` command

- Invokes `cc` adding all the necessary options (MPI libraries, path to `mpi.h`)
- Eases the compilation in different platforms
- `mpicc -show` shows the available options
- Also available: `mpicxx`, `mpif77`, `mpif90`

```
$ mpicc -o prgmpi prgmpi.c
```