CPA – Parallel Computing

Informatics Engineering Degree

S1. Introduction to Parallel Computing Frameworks

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

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Year 2016/17

etsinf

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Content

Section 1

Programming in "C"

- A brief remind on "C"

"C" Language

"C" is a general purpose programming language

- Features: compiled, portable and efficient
- Java and C++ inherit the sintax of "C"
- A simple language kernel, additional functionality by means of software libraries.
- One of the most used 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`
- Enumerated: `enum` (equivalent to an integer)
- Floating Point: `float`, `double`
- Void type: `void` (special usage)
- Derived type: `struct`, arrays, pointers

```
char c;
int i1,i2;
enum {NORTE,SUR,ESTE,OESTE} dir;
unsigned int k;
const float pi=3.141592;
double r=2.5,g;
```

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

New types can be defined using `typedef`

```
typedef enum {ROJO,VERDE,AZUL,AMARILLO,BLANCO,NEGRO} color;
color c1,c2;
```


Sentences and Expressions

There are different types of sentences:

- Declaration of types and variables (inside/outside of the function)
- Expression, typically an assignment `var=expr`
- Composed sentence (`{...}` block)
- Conditions (`if`, `switch`), loops (`for`, `while`, `do`)
- Others: void sentences (`;`), jump (`goto`)

Expressions:

- Assignations: `=`, `+=`, `-=`, `*=`, `/=`; increments: `++`, `--`
- Arithmetic: `+`, `-`, `*`, `/`, `%`; bit-wise: `~`, `&`, `|`, `^`, `<<`, `>>`
- Logic: `==`, `!=`, `<`, `>`, `<=`, `>=`, `||`, `&&`, `!`
  Zero is equivalent to "false" and any other value stands for "true"
- Ternary operator: `a? b: c`

Examples of flow control sentences

```
if (j>0) valor = 1.0;
else valor = -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 memory address of other variable or value

- 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 arithmetics

- Basic operations: `+`, `-`, `++`
- The increment/decrement is proportional to the pointed type size

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

NULL pointer

- It values zero (`NULL`)
- It is an invalid pointer 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)

Structs

Structs: 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 funtion type were `void`)

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

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

The arguments of a function receive the values indicated in the call (references can be achieved through 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)

Software 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`

Mathematic 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
- They are only accessible by the sentences of the function block
- They are created in the (*stack*), and destroyed when leaving the function

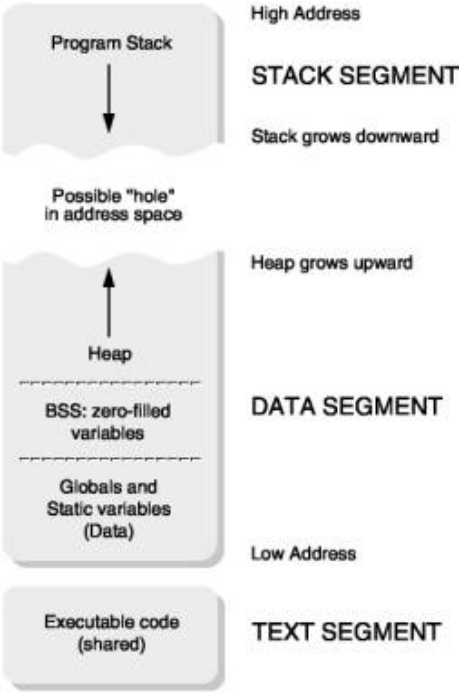
Static variables

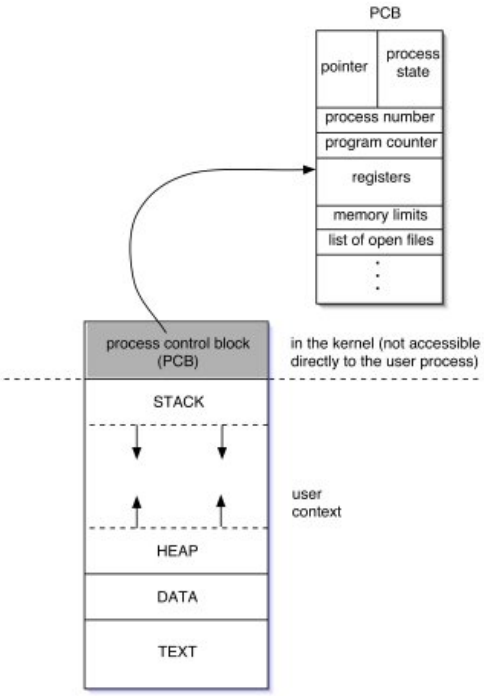
- Modifier `static`
- Local scope but persistent between successive calls

Dynamic memory allocated variables

- Variables pointing to memory allocated with `malloc`, persist until `free` function is called
- They are created in the *heap*

Memory model of Unix processes



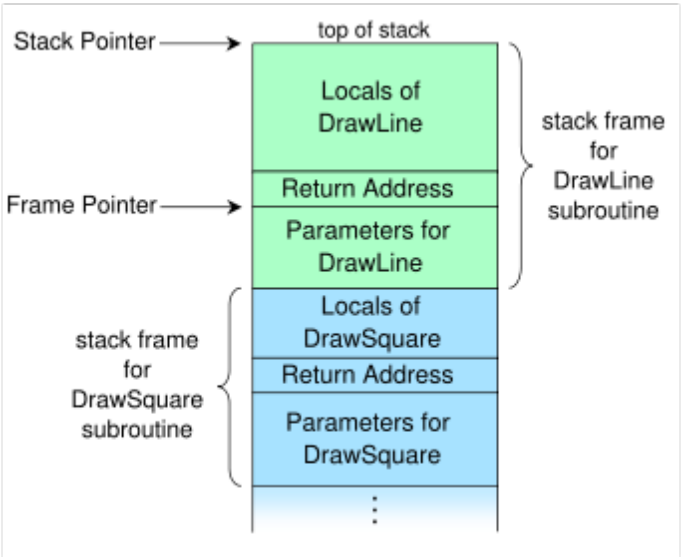


Stack

The arguments of a function are treated as local variables

When entering in 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
- Cluster for the laboratory
- Running parallel programs

Development cycle

The process of compiling consist of:

- Preprocessing: The "C" source code is modified according to a set of instructions (preprocessing directives)
- Compiling: 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 actions:

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

Preprocessing

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

- `include`: It inserts the content from other file
- `define`: It defines constants and macro-expressions (including arguments)
- `if`, `ifdef`: enables skiping part of the code during the 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
- It comprises the machine code of the functions and variables, as well as a list of unsolved symbols

Linking (*link*): `ld`

- It solves all the unsolved dependencies using the `*.o` files and external software libraries (`*.a`, `*.so`)

ej.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 ej ej.c f1.c -lm
```

Parallel program compilation

OpenMP is based on directives `#pragma omp`

- A compiler without OpenMP support ignores these directives
- Modern compilers have these support, provided that a special option is included in the compilation and linkage

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

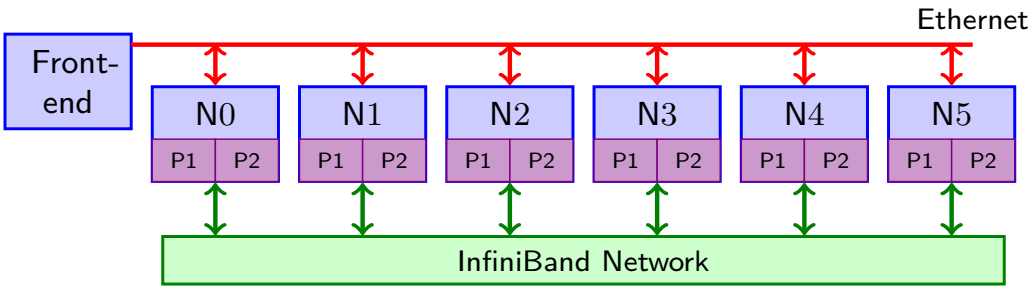
MPI provides the `mpicc` command

- Invokes `cc` adding all the necessary options for MPI codes (MPI software libraries, `mpi.h` path)
- Eases the compilation in different computers and environments
- `mpicc -show` shows the options available
- Also availbe for other languages: `mpicxx`, `mpif77`, `mpif90`

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


Cluster for the laboratory

Hardware configuration: 6 biprocessors linked with InfiniBand



Each node:

- 2 AMD Opteron 16 Core 6272 processor, 2.1Ghz, 16MB
- 32GB DDR3 1600 memory
- 500GB, SATA 6 GB/s HD
- InfiniBand QDR 40Gbps adapter

In total: 12 proc, 192 cores, 192 GB

Parallel computing: Front-End

The *front-end* enables users to interact with the cluster

Accessing the front-end:

```
$ ssh -X user@kahan.dsic.upv.es
```

For routine tasks. (Do not run costly executions)

- Editing and compiling programs
- Short test runs

Useful commands:

- Files/directories: `cd`, `pwd`, `ls`, `cp`, `mkdir`, `rm`, `mv`, `scp`, `less`, `cat`, `chmod`, `find`
- Processes: `w`, `kill`, `ps`, `top`
- Editors and others: `vim`, `emacs`, `pico`, `man`

Lab cluster: Network

Gigabit Ethernet

- Auxiliar network, only for the O.S. traffic (`ssh`, NFS)

InfiniBand

- High bandwidth network with low latency, perfect for clusters
- Evolution from other networks such as Myrinet
- Bidirectional bus, with serveral transfer rates (simple, doble, ...); links can be grouped by 4 or 12

	SDR	DDR	QDR
1X	2.5 Gbps	5 Gbps	10 Gbps
4X	10 Gbps	20 Gbps	**40 Gbps**
12X	30 Gbps	60 Gbps	120 Gbps

- Coding ratio 8B/10B: Effective communication bandwidth of 32 Gbps
- Latency: theoric 100ns, in practice around 1-2μs
- Switched topology (using a *switch*)

Running parallel programs

OpenMP: Executables can be run directly

Usually the number of threads should be indicated

```
$ OMP_NUM_THREADS=4 ./prgomp
```

Other option is to export the variables

```
$ OMP_NUM_THREADS=4; OMP_SCHEDULE=dynamic
$ export OMP_NUM_THREADS OMP_SCHEDULE
$ ./prgomp
```

MPI: use `mpiexec` (or `mpirun`) commands

Options: choose hosts, architecture

```
$ mpiexec -n 4 prgmpi <args>
$ mpiexec -n 6 -host node1,node2,node5 prgmpi
```

MPMD mode

```
$ mpiexec -n 2 program1 : -n 2 program2
```


Wueue system

The Queue system (or job scheduler or resource manager) is a software that enables sharing a cluster among different users

- The user can run "jobs" usually in *batch* mode (non interactive) using one or several nodes
- A job is a particular execution with a set of attributes (nodes, maximum execution time, etc.)
- Job scheduling policies are defined
- The system accounts the resources used (hours)
- Objective: maximize usage and minimize waiting time

Working procedure:

1. A job is defined and then submitted to a queue, which returns an identifier.
2. Depending on the workload, after a waiting time the job is run.
3. The output is obtained when the job finishes.

Lab cluster: Queues (1)

TORQUE is a queue system based on PBS (*Portable Batch System*)

Example of a job `prac1.sh`

```
#PBS -q cpa
#PBS -l nodes=1:ppn=8,walltime=2:00:00
cd $HOME/prac/p1
./imagenes
```

- `-q`: name of the queue to submit the job
- `-l`: list of resources (nodes, memory, architecture,...)
- `-N`: name of the job
- `-m`,`-M`: sends an e-mail
- `-j`: merges the standard output and error in a single file

For MPI, use `mpiexec` (`-n` is)

Lab cluster: Queues (2)

For the submission of a job:

```
$ qsub prac1.sh
3482.kahan
```

At the end, two files are created in the current directory: `prac1.o3482` (output) y `prac1.e3482` (error)

For browsing the status of the job:

```
$ qstat
Job id            Name      User    Time   S Queue
----------------- --------- ------- ------ - -----
3482.kahan        prac1.sh  alum1   0      Q cpa
```

Possible status: queued (`Q`), running (`R`), ended (`E`)

Job cancelling: `qdel`