CPA – Computación Paralela

Grado en Ingeniería Informática

S1. Introducción a los Entornos de Programación Paralela

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

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Curso 2020/21

etsinf

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Contenido

Apartado 1

Programación en C

- Recordatorio del Lenguaje C

El Lenguaje C

C es un lenguaje de programación de propósito general

- Características: compilado, portable y eficiente
- Java y C++ heredan la sintaxis de C
- Núcleo del lenguaje simple, funcionalidad extra mediante bibliotecas (librerías)
- Uno de los más utilizados en supercomputación

```
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 y Tipos Básicos

Todas las variables se han de declarar

- Enteros: `char`, `int`, `long`; modificador `unsigned`
- Enumerados: `enum` (equivale a un entero)
- Coma flotante: `float`, `double`
- Tipo vacío: `void` (uso especial)
- Tipos derivados: `struct`, arrays, punteros

```
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;
```

Se pueden definir nuevos tipos con `typedef`

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


Sentencias y Expresiones

Existen diferentes tipos de sentencias:

- Declaración de variables y tipos (dentro/fuera de función)
- Expresión, típicamente una asignación `var=expr`
- Sentencia compuesta (bloque `{...}`)
- Condicionales (`if`, `switch`), bucles (`for`, `while`, `do`)
- Otras: sentencia vacía (`;`), salto (`goto`)

Expresiones:

- Asignaciones: =, +=, −=, *=, /=; incrementos: ++, −−
- Aritméticas: +, −, *, /, %; a nivel de bit: ~, &, |, ^, <<, >>
- Lógicas: ==, !=, <, >, <=, >=, ||, &&, !
  El cero se asimila a "falso" y cualquier otro a "verdadero"
- Operador ternario: `a? b: c`

Ejemplos de Sentencias de Control de Flujo

```
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 NORTE:
    y += 1; break;
  case SUR:
    y -= 1; break;
  case ESTE:
    x += 1; break;
  case OESTE:
    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 y Punteros

Array: colección de variables del mismo tipo

- En la declaración se indica la longitud
- Los elementos se acceden con un índice (empieza en 0)

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

Arrays multidimensionales: `double matriz[N][M];`
Las cadenas son arrays de `char` acabadas con el carácter `'\0'`

Puntero: variable que contiene la dirección de otra variable

- En la declaración se añade `*` antes del nombre de variable
- El operador `&` devuelve la dirección de una variable
- El operador `*` permite acceder al dato apuntado

```
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] */
```

Más Sobre Punteros

Aritmética de punteros

- Operaciones básicas: +, −, ++
- El desplazamiento es del tipo al que apunta la variable

```
char s[] =
    "Comput. Paralela";
char *p = s;
while (*p!='P') p++;
```

Puntero nulo

- Su valor es cero (NULL)
- Se usa para indicar un fallo

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

Puntero genérico

- De tipo void*
- Puede apuntar a variables de cualquier tipo

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

Puntero múltiple: `double **p` (puntero a puntero)

Estructuras

Estructura: colección de datos heterogéneos

- Los miembros se acceden con . (o -> en el caso de puntero a estructura)

```
struct complejo {
  double re,im;
};
struct complejo 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";
```

Funciones

Un programa C se compone de al menos una función (`main`)

Devuelven un valor (a menos que la función sea de tipo `void`)

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

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

Paso de parámetros por valor (el paso por referencia se consigue mediante punteros)

```
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);
```

Se pueden declarar funciones antes de su definición (prototipo)

Funciones de Biblioteca

Operaciones de cadenas `<string.h>`

- Copia cadena (`strcpy`), compara cadena (`strcmp`)
- Copia memoria (`memcpy`), inicializa memoria (`memset`)

Entrada-salida `<stdio.h>`

- Estándar: `printf`, `scanf`
- Ficheros: `fopen`, `fclose`, `fprintf`, `fscanf`

Utilidades estándar `<stdlib.h>`

- Gestión de memoria dinámica: `malloc`, `free`
- Conversiones: `atof`, `atoi`

Funciones matemáticas `<math.h>`

- Funciones y operaciones: `sin`, `cos`, `exp`, `log`, `pow`, `sqrt`
- Redondeo: `floor`, `ceil`, `fabs`

Ejemplo con Fichero

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

void leedatos( 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);        /* numero de datos a cargar */
  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);
  procesa(n,ia,ja,va);
  free(ia); free(ja); free(va);
}
```


Tipos de Variables

Variables globales

- Se declaran fuera de cualquier función
- Acceso desde cualquier punto del programa
- Se crean en el segmento de datos

Variables locales

- Declaradas dentro de una función
- Visibles dentro del bloque
- Se crean en la pila (*stack*), se destruyen al salir

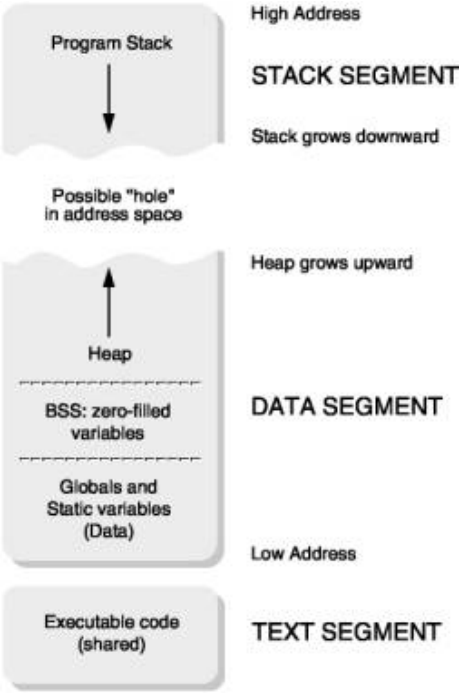
Variables estáticas

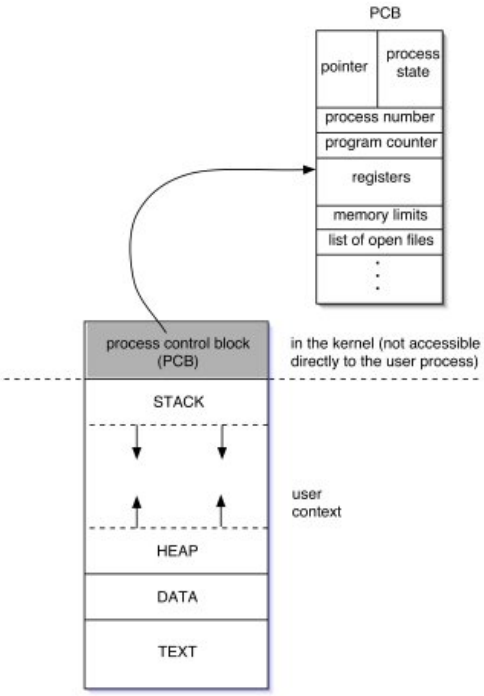
- Modificador `static`
- Ámbito local pero persistentes de una llamada a otra

Variables en memoria dinámica

- Memoria reservada con `malloc`, persisten hasta el `free`
- Se crean en el *heap*

Modelo de Memoria en Procesos Unix



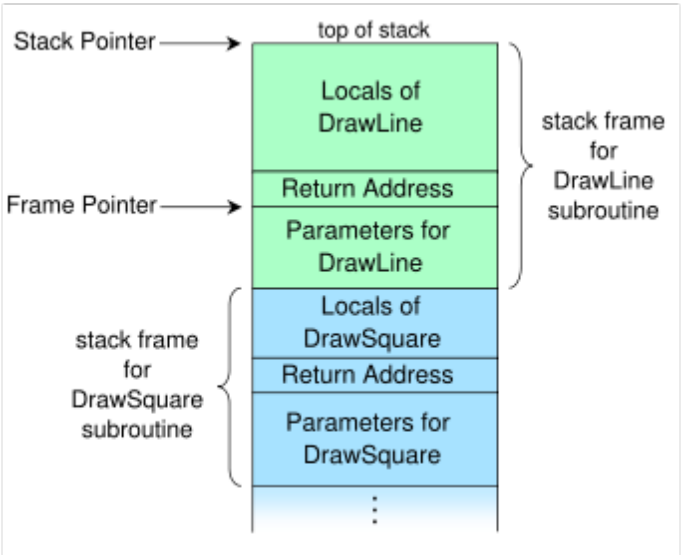


Pila de Llamadas

Los argumentos de una función son como variables locales

Al entrar en una función:

1. se apilan los argumentos
2. se apila la dirección de retorno
3. se crean las variables locales



Al hacer `return` se destruye todo el contexto creado

Apartado 2

Uso de Computadores Paralelos

- Ciclo de Desarrollo
- Cluster de Prácticas
- Ejecución de Programas Paralelos

Ciclo de Desarrollo

El proceso de compilación consta de:

- Preprocesado: modifica el código fuente en C según una serie de instrucciones (directivas de preprocesado)
- Compilación: genera el código objeto (binario) a partir del código ya preprocesado
- Enlazado: une los códigos objeto de los distintos módulos y bibliotecas externas para generar el ejecutable final

El ciclo de desarrollo se complementa con otros pasos:

- Automatizar compilación de programas complejos (`make`)
- Depuración de errores (`gdb`, `valgrind`)
- Análisis de prestaciones (`gprof`)

Preprocesado

Como paso previo a la compilación se hace el preprocesado (comando cpp, se invoca automáticamente)

- `include`: inserta el contenido de otro fichero
- `define`: define constantes y macros (con argumentos)
- `if`, `ifdef`: compilación condicional
- `pragma`: directiva de compilador

```
#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
```


Compilación y Enlazado

Compilación: `cc`

- Por cada fichero `*.c` se genera un `*.o`
- Contiene código máquina de las funciones y variables, y una lista de símbolos no resueltos

Enlazado o montaje (*link*): `ld`

- Resuelve todas las dependencias pendientes a partir de `*.o` y bibliotecas (`*.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
```

Compilación de Programas Paralelos

OpenMP se basa en directivas `#pragma omp`

- Un compilador sin soporte OpenMP ignora estas directivas
- Los compiladores recientes tienen soporte, con una opción (es necesaria tanto al compilar como al enlazar)

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

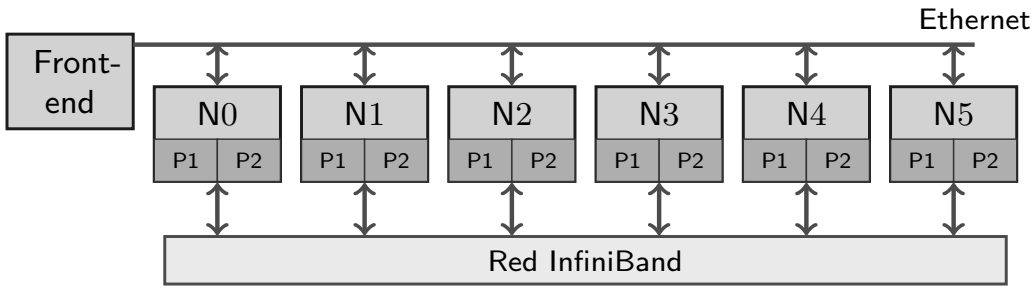
MPI proporciona el comando `mpicc`

- Invoca a `cc` añadiendo todas las opciones necesarias (bibliotecas de MPI, ruta de `mpi.h`)
- Facilita la compilación en diferentes máquinas
- `mpicc -show` muestra las opciones que se usarán
- También `mpicxx`, `mpif77`, `mpif90`

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


Cluster de Prácticas

Configuración hardware: 6 biprocesadores con red InfiniBand



Cada nodo:

- 2 proc. AMD Opteron 16 Core 6272, 2.1Ghz, 16MB
- 32GB de memoria DDR3 1600
- Disco 500GB, SATA 6 GB/s
- Controladora InfiniBand QDR 40Gbps

Agregado: 12 proc, 192 núcleos, 192 GB

Cluster de Prácticas: Front-End

El nodo cabecera (*front-end*) permite a los usuarios interactuar con el cluster

Conexión:

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

Para tareas rutinarias (no lanzar ejecuciones costosas)

- Edición y compilación de los programas
- Ejecuciones cortas para comprobar

Comandos útiles:

- Ficheros/directorios: `cd`, `pwd`, `ls`, `cp`, `mkdir`, `rm`, `mv`, `scp`, `less`, `cat`, `chmod`, `find`
- Procesos: `w`, `kill`, `ps`, `top`
- Editores y otros: `vim`, `emacs`, `pico`, `man`

Cluster de Prácticas: Red

Gigabit Ethernet

- Red auxiliar, sólo para tráfico del S.O. (`ssh`, NFS)

InfiniBand

- Red rápida de baja latencia, ideal para clusters
- Evolución de redes como Myrinet y otras
- Bus serie bidireccional, con varias tasas de transferencia (simple, doble, ...); los enlaces pueden agruparse en 4 o 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

- Codificación 8B/10B: tasa efectiva de 32 Gbps
- Latencia: teórica 100ns, real en torno a 1-2μs
- Topología conmutada (con *switch*)

Ejecución de Programas Paralelos

OpenMP: ejecutar directamente

Suele ser necesario indicar el número de hilos

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

Otra opción es exportar las variables

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

MPI: usar el comando `mpiexec` (o `mpirun`)

Opciones: seleccionar el host, la arquitectura

```
$ mpiexec -n 4 prgmpi <args>
$ mpiexec -n 6 -host nodo1,nodo2,nodo5 prgmpi
```


Sistemas de Colas

El sistema de colas (o planificador de trabajos o gestor de recursos) es un software que permite usar un cluster de forma compartida entre muchos usuarios

- El usuario puede lanzar "trabajos" normalmente en modo *batch* (no en interactivo) utilizando uno o más nodos
- Un trabajo (*job*) es una ejecución particular, con una serie de atributos (nodos, tiempo máximo de ejecución, etc.)
- Se definen políticas de planificación de trabajos
- El sistema contabiliza los recursos utilizados (horas)
- Objetivo: maximizar utilización, minimizar espera

Forma de trabajar:

1. Se define el trabajo y se lanza a la cola (da un identif.)
2. Tras un tiempo de espera, el trabajo se ejecuta
3. Al finalizar se recupera la salida producida

Cluster de Prácticas: Colas (1)

TORQUE es un sistema de colas abierto basado en PBS (*Portable Batch System*)

Ejemplo de trabajo `prac1.sh`

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

- `-q`: nombre de la cola a la que se lanza el trabajo
- `-l`: lista de recursos (nodos, memoria, arquitectura,...)
- `-N`: nombre del trabajo
- `-m,-M`: enviar un correo electrónico
- `-j`: juntar salida y error en el mismo fichero

Para MPI usar `mpiexec` (no hace falta `-n`)

Cluster de Prácticas: Colas (2)

Para lanzar:

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

Al terminar se crean en el directorio actual dos ficheros: `prac1.o3482` (salida) y `prac1.e3482` (error)

Para ver el estado:

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

Posibles estados: encolado (`Q`), ejecutando (`R`), terminando (`E`)

Cancelación de un trabajo: `qdel`