Parallel computing lecture

Part II. Reminder of the C programming language

Frédéric S. Masset

Instituto de Ciencias Físicas, UNAM

Outline

Outline

Outline

About tests and loops

In the first lecture we saw `if (test) {...}` (which can be `if (test) {...} else {...}`) and the loop `for (start; endtest; action) {...}`

There are some other useful test and loop commands. Here are the loop commands:

- `while (test) {...}` which executes what is inside `{...}` in loop, as long as test is true.
- `do {...} while (test);` which does the same, but it executes what is inside the `{...}` at least once.

About tests and loops

In the first lecture we saw `if (test) {...}` (which can be `if (test) {...} else {...}`) and the loop `for (start; endtest; action) {...}`

There are some other useful test and loop commands. Here are the loop commands:

- `while (test) {...}` which executes what is inside `{...}` in loop, as long as test is true.
- `do {...} while (test);` which does the same, but it executes what is inside the `{...}` at least once.

About tests and loops

In the first lecture we saw `if (test) {...}` (which can be `if (test) {...} else {...}`) and the loop `for (start; endtest; action) {...}`

There are some other useful test and loop commands. Here are the loop commands:

- `while (test) {...}` which executes what is inside `{...}` in loop, as long as test is true.
- `do {...} while (test);` which does the same, but it executes what is inside the `{...}` at least once.

additional test commands

The `switch` command allows to test the value of a variable in a variety of cases. Example:

```
switch (value)
   case 0:
       printf ("Value is zero\n");
       break;
  case 1:
       printf ("Value is one\n");
       break;
  case 2:
       printf ("Value is two\n");
       break;
  default:
       printf ("Value is something\n");
```

additional test commands

The `switch` command allows to test the value of a variable in a variety of cases. Example:

```
switch (value)
   case 0:
       printf ("Value is zero\n");  break required
       break;
  case 1:
       printf ("Value is one\n");
       break;
  case 2:
       printf ("Value is two\n");
       break;
  default:
       printf ("Value is something\n");
```

additional test commands

One-line quick test syntax:
We check a variable `b`. We want `a` to be one if `b` is negative, and 2 otherwise.

```
a = (b < 0 ? 1 : 2);
```

First case executed if test is true.
Second case executed if test is false.

Fast and efficient, but not extremely legible...

Outline

Structures in C

Assume that we want a function acting on an array that returns the mean and variance at the same time. How would we do that ?

```
struct pair {
    float mean;
    float var;
};

struct pair my_stat_func (float *array) {
...
}

int main () {
   struct pair result;
   ...
   result = my_stat_func (array);
   printf ("Average  = %f\n", result.mean);
   printf ("Variance = %f\n", result.var);
}
```

Structures in C

Assume that we want a function acting on an array that returns the mean and variance at the same time. How would we do that ?

```
struct pair {    structure : a "bag" that contains several things
    float mean;
    float var;
};

struct pair my_stat_func (float *array) {
...
}

int main () {
   struct pair result;
   ...
   result = my_stat_func (array);
   printf ("Average  = %f\n", result.mean);
   printf ("Variance = %f\n", result.var);
}
```

Structures in C

Assume that we want a function acting on an array that returns the mean and variance at the same time. How would we do that ?

```
struct pair {
    float mean;
    float var;
};
Here we have defined a new type: "struct pair"
struct pair my_stat_func (float *array) {
...
}

int main () {
   struct pair result;
   ...
   result = my_stat_func (array);
   printf ("Average  = %f\n", result.mean);
   printf ("Variance = %f\n", result.var);
}
```

Structures in C

Assume that we want a function acting on an array that returns the mean and variance at the same time. How would we do that ?

```
struct pair {
    float mean;
    float var;
};

struct pair my_stat_func (float *array) {
... This function returns a variable of type 'struct pair'
}

int main () {
   struct pair result;
   ...
   result = my_stat_func (array);
   printf ("Average  = %f\n", result.mean);
   printf ("Variance = %f\n", result.var);
}
```

Structures in C

Assume that we want a function acting on an array that returns the mean and variance at the same time. How would we do that ?

```
struct pair {
    float mean;
    float var;
};

struct pair my_stat_func (float *array) {
...
}

int main () {
   struct pair result;
   ...We access the components with a 'dot' and their names:
   result = my_stat_func (array);
   printf ("Average  = %f\n", result.mean);
   printf ("Variance = %f\n", result.var);
}
```

Pointers to structures

Now a structure may contain many things. Passing it by value to a function may be awkward. Furthermore, one may wish that the function modifies some fields of the structure...
We therefore need ... ?

Pointers to structures

Now a structure may contain many things. Passing it by value to a function may be awkward. Furthermore, one may wish that the function modifies some fields of the structure...
We therefore need a pointer to that structure.

```
void my_func (struct pair *my_pair) {
  ...
  average = (*my_pair).mean; //Notation quite heavy...
}
```

Pointers to structures

Now a structure may contain many things. Passing it by value to a function may be awkward. Furthermore, one may wish that the function modifies some fields of the structure...
We therefore need a pointer to that structure.

```
void my_func (struct pair *my_pair) {
  ...
  average = my_pair->mean; //Much better, and intuitive...
}
```

Pointers to structures

Now a structure may contain many things. Passing it by value to a function may be awkward. Furthermore, one may wish that the function modifies some fields of the structure...
We therefore need a pointer to that structure.

```
void my_func (struct pair *my_pair) {
  ...
  average = my_pair->mean; //Much better, and intuitive...
}
```

The notation −> allows to access the field of a structure handled by a pointer.

Outline

Comparison operators

Here is the list of the main comparison operators:

Notation	Meaning
$>$	Larger
$<$	Smaller
$<=$	Smaller or equal
$>=$	Larger or equal
$!=$	Different (! often means 'not' in C)
$==$	Equal (watch out !)

Comparison operators

Here is the list of the main comparison operators:

Notation	Meaning
$>$	Larger
$<$	Smaller
$<=$	Smaller or equal
$>=$	Larger or equal
$!=$	Different (! often means ‘not’ in C)
$==$	Equal (watch out !)

Comparison operators

Here is the list of the main comparison operators:

Notation	Meaning
$>$	Larger
$<$	Smaller
$<=$	Smaller or equal
$>=$	Larger or equal
$!=$	Different (! often means 'not' in C)
$==$	Equal (watch out !)

Comparison operators

Here is the list of the main comparison operators:

Notation	Meaning
$>$	Larger
$<$	Smaller
$<=$	Smaller or equal
$>=$	Larger or equal
$!=$	Different (! often means 'not' in C)
$==$	Equal (watch out !)

Comparison operators

Here is the list of the main comparison operators:

Notation	Meaning
$>$	Larger
$<$	Smaller
$<=$	Smaller or equal
$>=$	Larger or equal
$!=$	Different (! often means 'not' in C)
$==$	Equal (watch out !)

Comparison operators

Here is the list of the main comparison operators:

Notation	Meaning
$>$	Larger
$<$	Smaller
$<=$	Smaller or equal
$>=$	Larger or equal
$!=$	Different (! often means 'not' in C)
$==$	Equal (watch out !)

Comparison operators

A common mistake

```
int b=3, a=1;
if (a = b)
   printf ("a and b are equal\n");
else
   printf ("a and b are different\n");
```

What do you expect ?

Logical operators

Notation	Meaning
&&	and
||	or
!	not

Example: what does the following test mean ?

```
if (!((a == b) || (b == c)))
```

Logical operators

Notation	Meaning
&&	and
||	or
!	not

Example: what does the following test mean ?

```
if (!((a == b) || (b == c)))
```

Logical operators

Notation	Meaning
&&	and
||	or
!	not

Example: what does the following test mean ?

```
if (!((a == b) || (b == c)))
```

Outline

What is a string in C ?

A string is an array of characters (byte). Technically, it is a pointer to the first element of the string:

```
char foo[100] = "This is an interesting string";
char aaa[100] = "That is an interesting string";
foo[2] = 'a';
foo[3] = 't';
printf ("%s\n", foo);
```

What do you expect ?

What is a string in C ?

A string is an array of characters (byte). Technically, it is a pointer to the first element of the string:

```
char foo[100] = "This is an interesting string";
char aaa[100] = "That is an interesting string";
foo[2] = 'a';
foo[3] = 't';
if (aaa == foo) printf ("Strings are identical\n");
```

What do you expect ?

What is a string in C ?

A string is an array of characters (byte). Technically, it is a pointer to the first element of the string:

```
char foo[100] = "This is an interesting string";
char aaa[100] = "That is an interesting string";
foo[2] = 'a';
foo[3] = 't';
if (aaa == foo) printf ("Strings are identical\n");
```

What do you expect ?
String manipulation in C is a bit awkward. Here you must use `strcmp` or variants.

Build, run and time the example program
`examples/matrix_sum.c`
Now change the loop order and retry. What do you notice ?

This is rule number 1, even before anything you will learn about parallel programming: think of the cache, always !

Build, run and time the example program
`examples/matrix_sum.c`
Now change the loop order and retry. What do you notice ?

This is rule number 1, even before anything you will learn about parallel programming: think of the cache, always !