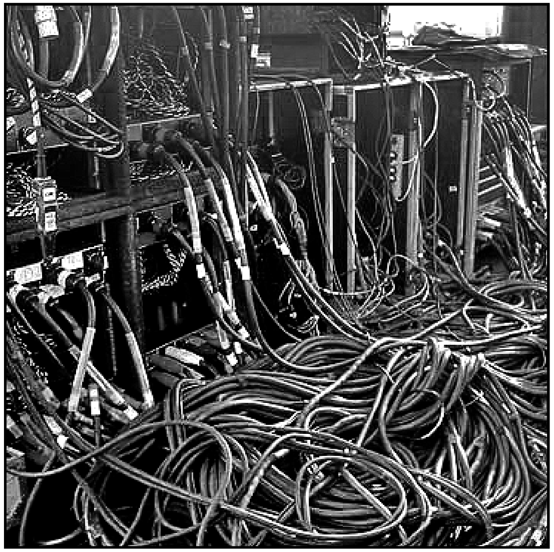
Concurrent Computing

Lecturers: Prof. Majid Mirmehdi (majid@cs.bris.ac.uk)
Dr. Tilo Burghardt (tilo@cs.bris.ac.uk)
Dr. Daniel Page (page@cs.bris.ac.uk)

Web: http://www.cs.bris.ac.uk/Teaching/Resources/COMS20001



LECTURE 17

ADVANCED DATA ACCESS IN xC

Recap: Distributables

```
//distributable.xc
#include <platform.h>
#include <stdio.h>
#include <string.h>

//define a communication interface i
typedef interface i {
  int f(int a[]); void g(); } i;

//server tasks providing functionality are ONLY responsive
[[distributable]] //allows on-demand core allocation to caller core
void myServer(server i myInterface[n], unsigned n, int index) {
  while (1)
    select {
      case myInterface[int j].f(int a[]) -> int returnval:
        printf("f called from i%d-c%d \n",index,j);
        returnval = a[j]+j;
        break;
      case myInterface[int j].g():
        printf("g was called from i%d-c%d\n",index,j);
        break;
} }

//client task calling functions
void myClient(client i myInterface, int j) {
  int a[2] = {1+j,2+j};
  printf("Client %d received %d\n",j,myInterface.f(a));
  if (j%2==0) myInterface.g();
}

//main starting six threads
int main() {
  interface i myInterface[2];
  interface i myInterface1[2];
  par { //uses only 4 cores
    myClient(myInterface[0],0);
    myClient(myInterface[1],1);
    myClient(myInterface1[0],2);
    myClient(myInterface1[1],3);
    myServer(myInterface,2,0);
    myServer(myInterface1,2,1);
  }
  return 0;
}
```

instructs compiler only to allocate core in case function is triggered/called (core of caller can be utilised for this, if on same tile)

Distributables must be:

1) functions that satisfy combinable criteria

2) cases within the **select** only respond to interface calls

```
f called from i1-c1
f called from i0-c1
f called from i1-c0
f called from i0-c0
Client 3 received 6
Client 1 received 4
Client 2 received 3
Client 0 received 1
g was called from i1-c0
g was called from i0-c0
```

Recap: References

```
//references.xc
#include <platform.h>
#include <stdio.h>
#include <string.h>

//function with callByValue parameter
void f(int callByValue) { callByValue++; }

//function with callByReference parameter
void g(int &callByRef) { callByRef++; }

//thread for testing
void taskA(chanend c, int index) {
  int var[5] = {1,2,3,4,5};
  //making references
  int &ref = var[index];
  printf("T%d var:%d, ref:%d\n",index,var[index],ref);
  //changing variable via their reference
  ref = 6+index;
  printf("T%d var:%d, ref:%d\n",index,var[index],ref);
  //using reference as callByValue parameter
  f(ref);
  printf("T%d var:%d, ref:%d\n",index,var[index],ref);
  //using/changing reference as callByReference parameter
  g(ref);
  printf("T%d var:%d, ref:%d\n",index,var[index],ref);
}

//main starting threads
int main() {
  chan c;
  par {
    on tile[0]:taskA(c,1);
    on tile[1]:taskA(c,0);
  }
  return 0;
}
```

```
T1 var:2, ref:2
T0 var:1, ref:1
T1 var:7, ref:7
T0 var:6, ref:6
T1 var:7, ref:7
T0 var:6, ref:6
T1 var:8, ref:8
T0 var:7, ref:7
```

The programmer can create references to variables, this includes elements of arrays.

Manipulating the value of references (or aliases) will manipulate the value of the referenced variable and vice versa.

Function arguments can be references. This is known as call-by-reference.

Recap: Array References

```
//dataexchange2.xc
#include <platform.h>
#include <stdio.h>

//define a communication interface i
typedef interface i {
  int f(int a[]);
  void g();
} i;

//server task providing functionality of i
void myServer(server i myInterface) {
  int serving = 1;
  while (serving)
    select {
      case myInterface.f(int a[]) -> int returnval:
        printf("f receives: %d \n", a[0],a[1]);
        a[1] = a[0]*2;
        returnval = a[0];
        break;
      case myInterface.g():
        printf("g was called\n");
        serving = 0;
        break;
} }

//client task calling functions
void myClient(client i myInterface) {
  int a[2] = {1,0};
  printf("f returns: %d \n", myInterface.f(a));
  printf("a[1] set to: %d \n", a[1]);
  myInterface.g();
}

//main starting two threads
int main() {
  interface i myInterface;
  par {
    on tile[0]: myServer(myInterface);
    on tile[1]: myClient(myInterface);
  }
  return 0;
}
```

```
f receives: 1
f returns: 1
a[1] set to: 2
g was called
```

Arrays are naturally passed as references.

References can be passed between tasks as interface function parameters.

The function f in the interface example can alter the array a provided as an argument.

Manipulations across tiles are performed by communication between the threads over the on-chip network.

Pointer Aliasing

```
//pointers1.xc
#include <platform.h>
#include <stdio.h>

void taskA() {
  int var = 1;
  int var2 = 1;
  int &ref = var;
  int *alias pointer = &var;
  int *restrict limited = &var2;
  //cannot alias, reassign or copy limted or
  //use var2 itself ever again
  printf("Var = %d",var);
  //printf("Var2 = %d",var2); produces error
}

int main() {
  taskA();
  return 0;
}
```

Var = 1

In xC, no task is allowed to access memory that another task owns.

Memory ownership can transfer between threads at well-defined points in the program.

To facilitate this every pointer in xC is allocated a kind:

1) Restricted

2) Aliasing

3) Movable

4) Unsafe

Restricted Pointers – Common Pitfalls 1...

```
//pointers3.xc
#include <platform.h>
#include <stdio.h>

int var = 1;

void function(int *restrict p) {
  var = 5;
}

void taskA() {
  function(&var);
  int &ref = var;
}

int main() {
  taskA();
  return 0;
}
```

Spot the Compilation Error

```
//pointers4.xc
#include <platform.h>
#include <stdio.h>

int var = 1;
int var2 = 2;

void function(int *p) {
  p = &var2;
}

void taskA() {
  function(&var);
  int &ref = var;
}

int main() {
  taskA();
  return 0;
}
```

Spot the Compilation Error

Restricted Pointers – Common Pitfalls 2…

```
//pointers5.xc
#include <platform.h>
#include <stdio.h>

int var = 1;

void function(int *p, int *q) {
}

void taskA() {
  function(&var,&var);
}

int main() {
  taskA();
  return 0;
}
```

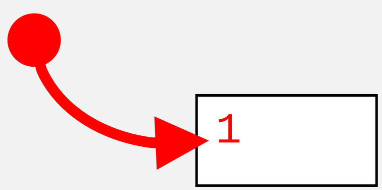
Spot the Compilation Error

```
//pointers6.xc
#include <platform.h>
#include <stdio.h>

int var = 1;
int var2 = 2;

void function(int *p) {
  int *restrict q;
  q = p;
}

void taskA() {
  function(&var);
  int &ref = var;
}

int main() {
  taskA();
  return 0;
}
```

Spot the Compilation Error

Pointer Parameters 1

```
//pointers7.xc
#include <platform.h>
#include <stdio.h>

int var = 1;

void function(int *alias p) {
  int *pointing = p;
  int *pointing2 = pointing;
  p = pointing2;
  printf("%d",*p);
}

void taskA() {
  int *restrict pointer = &var;
  function(pointer); //allowed
}

int main() {
  taskA();
  return 0;
}
```



In xC, no task is allowed to access memory that another task owns.

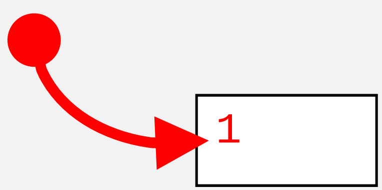
The programmer cannot pass alias pointers to different tasks in parallel (e.g. using the par statement).

The programmer also cannot have indirect access to an alias pointer such as a pointer to an alias pointer.

However, restricted pointers can be passed to an alias function parameter. The pointer Is then locally dealt with as an alias pointer.

Pointer Parameters 2

```
//pointers8.xc
#include <platform.h>
#include <stdio.h>

void function( int *p1, int *p2) {
  printf("%d",*p1);
}

void taskA() {
  int i = 1, j = 2;
  int *alias p = &i;
  int *alias q = &j;
  function(p, q); //valid, p no alias of q
}

int main() {
  taskA();
  return 0;
}
```



In xC, an aliasing pointer can be passed to a function taking a restricted pointer if it does not alias any other arguments.

The source of the aliasing pointer, however, has to be local to the calling function.

If it is from an incoming argument or global variable, an additional restriction is made: the function being called cannot access any global variables.

Pointer Parameters – Common Pitfalls...

```
//pointers8.xc
#include <platform.h>
#include <stdio.h>

int global = 5;

int function(int *q) {
  return *q + global;
}

void taskA() {
  int *alias p = &global;
  function(p);
}

int main() {
  taskA();
  return 0;
}
```

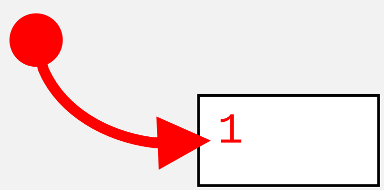
Spot the Compilation Error

```
//pointers9.xc
#include <platform.h>
#include <stdio.h>

int global = 5;

int function(int *q, int var) {
  return *q + global;
}

void taskA() {
  int var;
  int *restrict p = &var;
  function(p,var);
}

int main() {
  taskA();
  return 0;
}
```

Spot the Compilation Error

Movable Pointers

```
//pointers10.xc
#include <platform.h>
#include <stdio.h>

int * movable global;

void function(int * movable p) {
  global = move (p);
}

int * movable function2( void ) {
  return move (global);
}

void taskA() {
  int i = 1;
  int * movable p = &i;
  function(move(p));
  p = function2();
  printf("%d",*p);
}

int main() {
  taskA();
  return 0;
}
```



In xC, pointers that are restricted, but can be changing ownership are called movable.

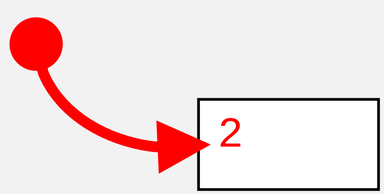
This allows access in different threads or access to global pointers across different scopes.

The move operator has to be used when transferring ownership, passing a movable pointer into a function or returning a movable pointer.

Movable pointers must point to the same location they were initialized with when going out of scope.

Changing Ownership

```
//pointers11.xc
#include <platform.h>
#include <stdio.h>

int j = 2;
int * movable q;
int * movable p = &j;

interface inf {
  void function(int * movable p);
};

void taskA(server interface inf i) {
  select {
    case i.function(int * movable p):
    q = move (p); //ownership is transferred
    break ;
  }
  printf ("%d", *q); //owning it here
}

void taskB(client interface inf i, int * movable p) {
  i.function( move(p));
}

int main() {
  interface inf i;
  par {
    taskA(i);
    taskB(i,move(p));
  }
  return 0;
}
```



In xC, to transfer a pointer beyond the scope of a transaction, movable pointers can be used.

By handing over ownership, one task must give up control of the memory region at a well defined point for the other task to use it.

This helps to avoid accidental race conditions.

Unsafe Pointers and Regions

```
//pointers12.xc
#include <platform.h>
#include <stdio.h>

int j = 2;

interface inf {
  void function(int *unsafe p);
};

unsafe void taskA(server interface inf i) {
  select {
    case i.function(int *unsafe p):
    printf ("%d\n", *p); //race condition applies!
    break ;
  }
}

unsafe void taskB(client interface inf i, int * unsafe p) {
  i.function(p);
  *p = 3; //truly GLOBAL access to j now...
}

unsafe int main() {
  interface inf i, k;
  int *unsafe p = &j;
  par {
    taskA(i);
    taskB(i,p);
    taskA(k);
    taskB(k,p);
  }
  return 0;
}
```

Declaration location	Default
Global variable	Restricted
Parameter	Restricted
Local variable	Aliasing
Function returns	No default

From xC Programming Guide

In xC, an unsafe pointer is provided for compatibility with C.

The programmer has to ensure memory safety for this type.

An unsafe pointer is not visible unless accessed in an unsafe region.

A function or block can be marked as unsafe to show that its body is an unsafe region.