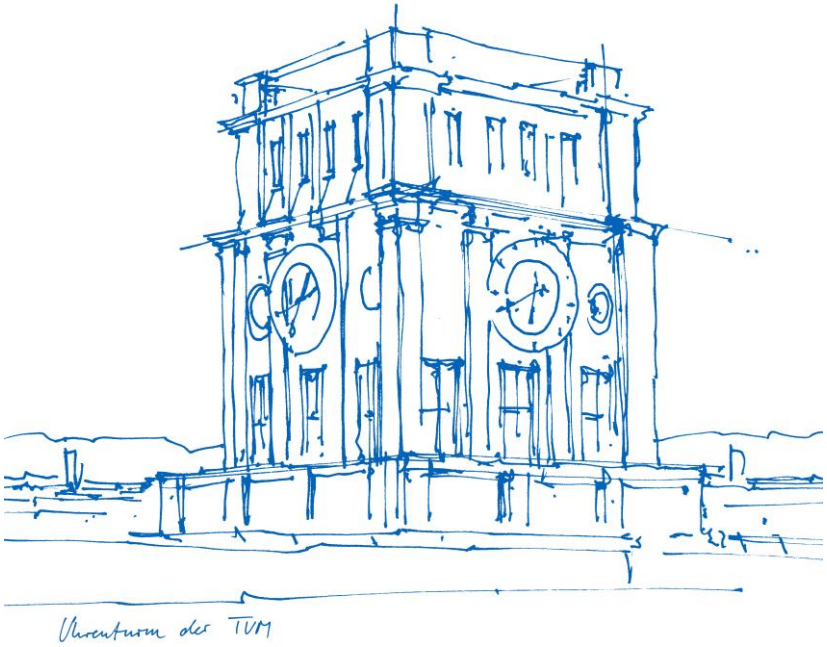
Future Trends in Computing
X10 Basics – Parallel Programming

Alexander Pöppl, Emily Ao Mo-Hellenbrand

Technical University of Munich

Department of Informatics

Chair of Scientific Computing in Computer Science



Activities

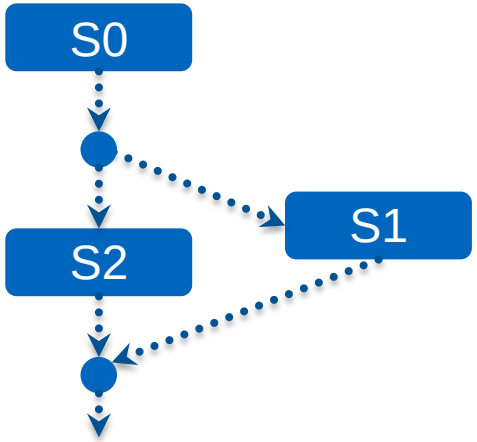
In X10, the fundamental unit of parallelism is the **Activity** (Very light-weight, X10-level thread).

Activities are executed concurrently by a number of worker threads in the X10 runtime.

May be used for short-lived actions (incrementation of variables) or for long-running functionality such as the run loop of a simulation.

Control flow is possible, however, no direct manipulation from outside (i.e. no interrupting, blocking, resuming or killing activities).

```
S0;
async S1;
S2;
```



Activities – Termination

Consider example:

```
async {s1();}
async {s2();}
```

What happens?

Local vs. Global Termination

- An activity terminates locally, once all its computation is finished
  - Root activity, after two child-activities are started
  - Its children once s1 and s2 are finished
- An activity terminates globally, once it is terminated locally itself and all its child activities are (recursively) globally terminated
  - The children terminate once all activities spawned in s1 (or s2) are globally terminated and the execution of s1 (or s2) is finished
  - The root activity terminates globally once the two children are spawned and terminated globally

Finish

Wait for the termination of all activities spawned in S

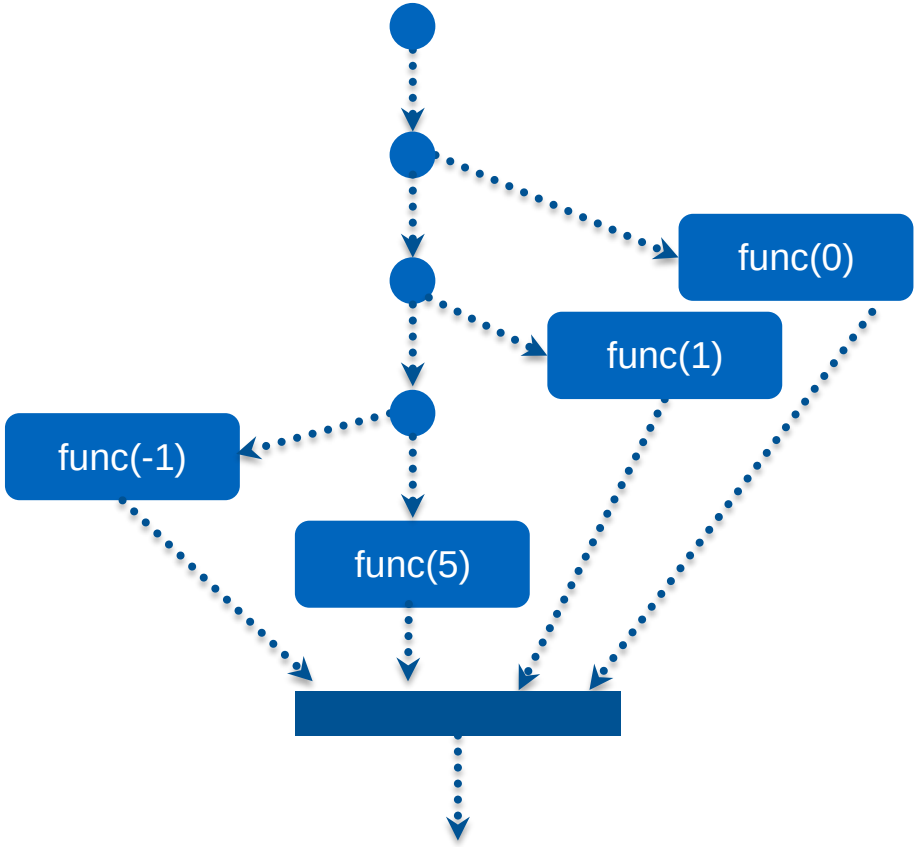
```
finish S;
```

Examples:

```
finish {
  async {s1();}
  async {s1();}
}

finish for (i in 0..3) async {
  func(i);
}
```

Finish

```
finish {
  for (i in 0..1) async {
    func(i);
  }
  async func(-1);
  func(5);
}
```



Atomic Blocks

What happens?

```
val x = [1,2,3,4];
async x(0) = x(0) + 4;
x(0) = (x(0) < 2) ? -x(0) : x(0);
```

Atomic blocks (**atomic** S) or (**when** (c) S) provide functionality to coordinate access to shared data.

Atomic blocks are executed atomically (i.e. in one step) in respect to each other.

```
val x = [1,2,3,4];
async atomic x(0) = x(0) + 4;
atomic x(0) = (x(0) < 2) ? -x(0) : x(0);
```

Are there still race conditions?

Atomic – Limitations

There are limitations to code S executed within an atomic:

- S may not spawn other activities
- S may not use blocking statements such as when, Clock.advanceAll(), finish
- S may not shift between places.
- ➢ S must be *sequential*, *single-place* and *non-blocking*

Atomic blocks are only atomic in regards to each other, and to nothing else.

```
var n : Long = 0;
finish {
  async atomic n = n + 1;
  async atomic n = n + 2;
}
```

```
var n : Long = 0;
finish {
  async n = n + 1;
  async atomic n = n + 2;
}
```

Rule of thumb: If one access to a variable is performed atomically, then all the others should be made atomic as well.

Atomic imposes a place-wide lock, more fine-granular locking is available.

Conditional Atomic Blocks

Written as **when** (c) S, conditional atomics execute S once condition c is met. For S, the same restrictions as for code in atomic blocks apply. The guard c should not have side effects.

```
val l = new ArrayList[String]();
async {
  // Do something...
  when(!l.isEmpty()) {
    val element = l.removeFirst();
  }
}
//Do something else...
atomic l.add("Some random element");
```

When is only guaranteed to be alerted if the state of an affected variable is changed in an atomic block, otherwise it may or may not be alerted.

Clocks

A significant number of parallel algorithms are executed in phases: Before proceeding with the execution of the next phase, all activities participating in the computation have to finish the current phase.

Clocks are created using the Clock.make() factory method.

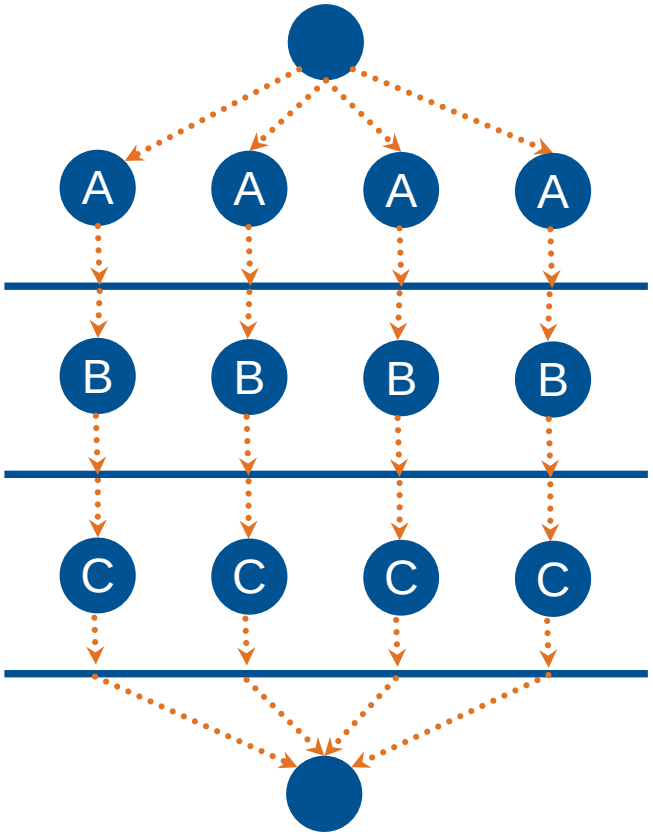
The following operations (only from activities that are registered to the clock already) are offered by a Clock c:

- Spawning a new activity that is immediately registered on the clock c: async clocked (c) S
- Unregistering the current activity from c: c.drop()
- Resuming the clock using c.resume(), an indication that the activity is finished with the current phase
- Waiting on the other registered activities using c.wait(). Blocks until all registered activities have called c.resume().

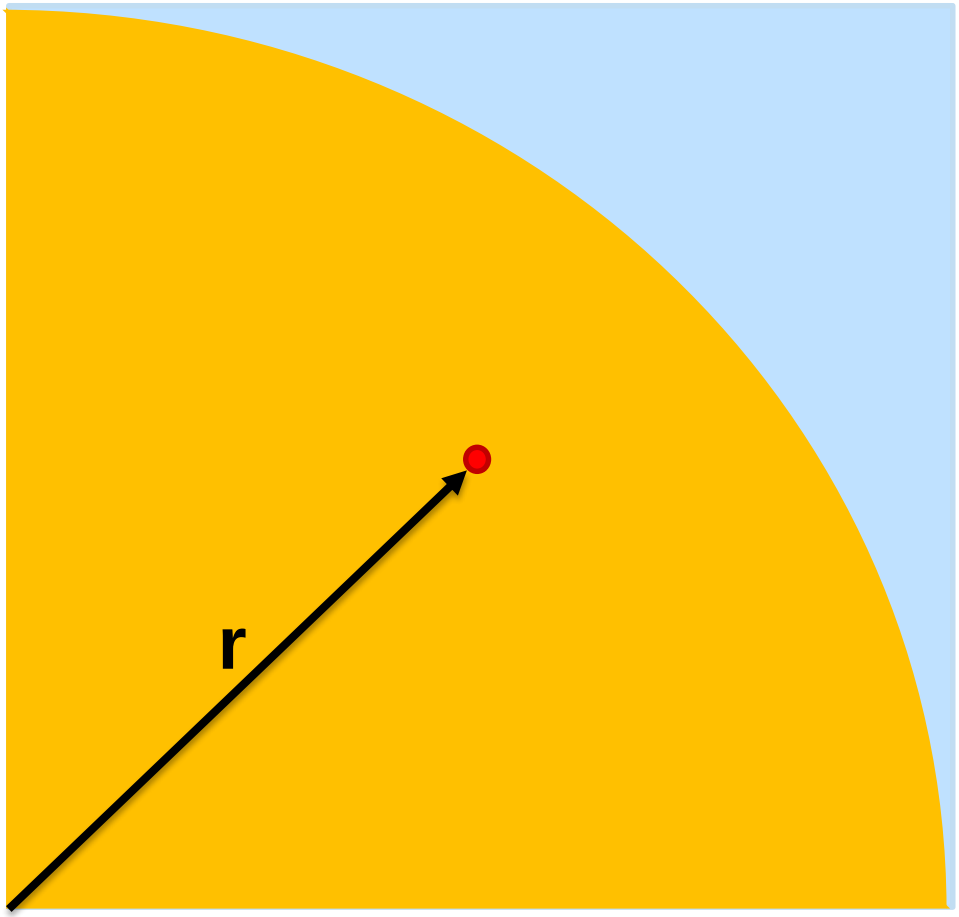
```
// ACTIVITY a
val c = Clock.make();
c.resume();

async clocked (c) {
  // ACTIVITY b
  c.advance();
  b_phase_two();
  // END OF ACTIVITY b
}

c.advance();
a_phase_two();
// END OF ACTIVITY a
```

Clocks – Shorthand Notation

```
clocked finish for (i in 0..3) clocked async {
  phase("A", i); // A
  Clock.advanceAll();
  phase("B", i); // B
  Clock.advanceAll();
  phase("C", i); // C
  Clock.advanceAll();
}
```



Exercise: Monte Carlo Pi



(r > 1) ? Inside : Outside

$$\text{Pi} = \frac{\#\ samples\ inside}{Total\ \#\ samples}$$

Exercise: Monte Carlo Pi

Implement a class **MontyPi** that calculates the value of **Pi** using the above described algorithm.

Make use of the X10 parallel features such that your program can be executed on a shared-memory architecture.

Note: Be careful with concurrent writes to the same address space.

Exercise: Heat Simulation

- Simulation of the temperature in a 2D plate $[0,1] \times [0,1]$.

- Governing PDE: ***Poisson's Equation***

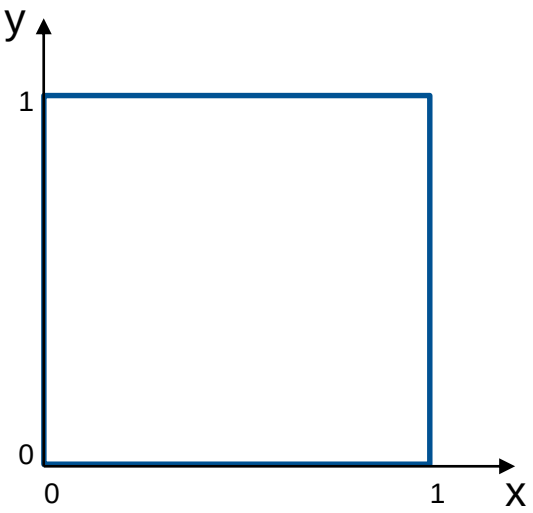
$$\boldsymbol{u_t = \Delta u + s(x,y)}$$

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + s(x,y)$$

  - **u**: the temperate at point (x,y) at time t
  - $s(x,y) = 0$
  - Initial condition:

    $\boldsymbol{u(x,y,0) = 1.0}$ for all $\boldsymbol{(x,y)}$
  - Boundary condition (Dirichlet):

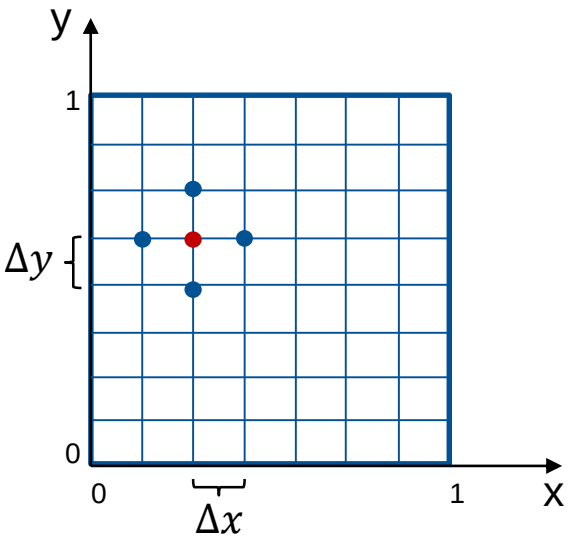
    $\boldsymbol{u(x,y,t) = 0}$ for all $\boldsymbol{(x,y) \in]0;1[^2}$



Exercise: Heat Simulation

1. Spacial discretization:
   discretize the continuous domain with a Cartesian grid

$$u_{i,j} = u(x, y), \qquad x = i\Delta x, \qquad y = j\Delta y$$



2. Time discretization: **Explicit Euler scheme**
   the temperature at the next time step $t_{n+1} = t_n + \Delta t$ is calcuated as:

$$u_{i,j}^{n+1} = u_{i,j}^{n} + \Delta t \frac{\partial u_{i,j}^{n}}{\partial t}$$

3. Laplacian operator Δ approximation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2}$$

4. Stability condition: $\frac{\Delta t}{\Delta x^2} + \frac{\Delta t}{\Delta y^2} \leq \frac{1}{2}$

Exercise: Heat Simulation

```
Nx = … // user specify simulation resolution
Ny = … // user specify simulation resolution
Sim_time = … // user specify simulation time

dx = …
dy = …
dt = …
u = [1.0, 1.0, …]  // initialize vector u
t = 0

while (t ≤ Sim_time) do
        for all u[i,j] do
                u_next[i,j] = a*u[i-1,j] + b*u[i+1,j] + c*u[i,j] + d*u[i,j-1] + e*u[i,j+1]
        t += dt
        u = u_next

return u
```

Exercise: Heat Simulation

Implement a class **HeatSim** that computes the temperature of a 2D plate using the above given conditions

Make use of the X10 parallel (shared-memory) features, Regions, Points, etc.