Australian National University

IMPLICIT CONCURRENCY

Week 8 Laboratory for Concurrent and Distributed Systems

Uwe R. Zimmer

Pre-Laboratory Checklist

- ❑ **You have read this text before you come to your lab session.**
- ❑ **You understand and can utilize message passing.**
- ❑ **You have a firm understanding of memory based synchronization.**
- ❑ **You can create and control tasks.**

Objectives

So far you saw many ways to reflect a given problem in a concurrent implementation. You did so by thinking about how multiple tasks could be utilized and how they would communicate and coordinate to achieve the set goal. This lab looks into the options of gaining the same effect without ever mentioning tasks in your code.

Interlude: Data parallelism

Did you ever wonder how repetitive most of the vector and matrix operations appear to be? Look for instance at this vector operation:

$$a \cdot \vec{v} = a \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a \cdot x \\ a \cdot y \\ a \cdot z \end{pmatrix}$$

Clearly, all three scalar-arithmetic operations are independent and could be done in parallel. Applying an operation to all element of a data structure is usually referred to as **mapping**. In the context of *category theory* and especially if this mapping changes the type of the data-structure, this is often also referred to as a **functor** (for the mathematically inclined).

What happens here?

$$\vec{v_1} + \vec{v_2} = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} + \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} x_1 + x_2 \\ y_1 + y_2 \\ z_1 + z_2 \end{pmatrix}$$

In this case we combine data structures, which is usually referred to as **zipping**. Again the individual operations are independent.

This is nice, but is there much of an advantage of doing these things in parallel? In terms of computational complexity we would reduce a $\Theta(n)$ operation to a $\Theta(1)$ operation. This sounds good, but we neglect the overhead which comes with distributing the data to different computing nodes and recollecting the results.

The actual benefit (if any) for a 3 dimensional vector is negligible, but think that data vectors are not necessarily 3 dimensional but scale up to thousand or millions of entries. If you can still keep the transition from $\Theta(n)$ operation to a $\Theta(1)$ there will be a significant change in performance.

Building a CRAY-1 at United Computer Systems 1979

Making those operations easier in order to gain the full benefit from the $\Theta(1)$ complexity was a major driver for the super-computer development from the 1970's to the late 90's.

The first computer which utilized this concept to drive performance into new dimensions was the Cray-1, which was the first commercial product by Cray (the super-computer company founded by Seymour Cray). Fujitsu, Hitachi and NEC followed with similar designs a few years later (… yet none had this iconic and memorable shape of the Cray-1, which was associated with super-computing in general for decades in popular culture).

Higher degrees of integration led to vector processor units which migrated into CPU's by the late 90's. You may recognize the names AltiVec, MMX or SSE. All of those are miniature versions of the original vector processing concept. Graphics processing units (GPUs) are another variation on the theme as applying the same operation concurrently to many pixels is nothing but a vector operation as well.

In practice it turns out that completely independent operations are not addressing all needs. Look for instance at this expression:

$$\vec{v_1} = \vec{v_2} \Rightarrow \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} = \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} \Rightarrow (x_1 = x_2) \wedge (y_1 = y_2) \wedge (z_1 = z_2)$$

It also has independent expressions, but then it has to be **reduced** to a single scalar. This is often referred to as **folding**. You are familiar with the concept from functional programming as well as from the "Reduce" exercise from lab 6.

But now to the actual topic of this interlude:

How to express such operations in code?

So far, you branched off into multiple tasks, combined results and re-synchronized for the result. And you did all of that by means of "explicit tasking", meaning you programmed how many tasks there should be and how they should communicate or synchronize.

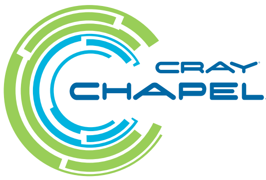
Lawrence Livermore National Laboratory, 1978

Are there other options?

For once you could not do any of it and hope that your compiler will recognize what you wanted to do, and translates those code sections automatically into concurrent versions.

Titan, 2012

This indeed happens to a degree and with some compilers and languages, but there is a better option: You use a language which can express data-parallel constructs. This way the compiler does not need to guess what this code is supposed to do, and you don't need to explicitly write out how things are translating



into concurrent entities which make maximal use of the available hardware (because this is something your compiler does actually know about).

Have a look at those four lines of Chapel code:

```
const Vector = {1 .. 100000000};

const Vector_1,
      Vector_2 : [Vector] real,
      Addition : [Vector] real = Vector_1 + Vector_2;
```

You may now suspect that Chapel has some vector library and the + is overloaded to work on those vectors. This is indeed not what happens here. The + is a + which works only on two scalar real values. The semantic behind this expression is different: The scalar operator, when applied to a data structure like an array is **promoted** to be applied component-wise between the two structured values. In a functional programming language you would probably say: The two vectors are automatically **zipped** together with the + operation:

```
zipWith (+) vector_1 vector_2
```

And here is where it becomes interesting:

The + operator is side-effect free, so the order in which it is applied to the components of the two vectors is irrelevant – including concurrent application of the + operation. The compiler can now chose freely how to employ the available hardware to run this program. While this could happen in any pure functional programming language (but unfortunately still doesn't in the main releases of those compilers), it will actually happen in Chapel. If you run those few lines of Chapel code above, expect to see all your CPU cores in action – without you ever mentioning any tasks or CPU cores in your source code.

The same rationale applies if you map an operation over a data-structure:

```
map (scale *) vector_1
```

or in Chapel:

```
const Scale  : real          = 5.1,
      Scaled : [Vector] real = Scale * Vector_1;
```

Again the scalar * operator is **promoted** to be applied to the vector value component-wise. And as well, the compilers are free to employ any concurrent hardware available – which in the case of Chapel also happens.

How about this one:

```
while Vector_1 != Vector_2 do {
  ...
}
```

(!= is the Chapel operator for "not equal".) Following the theme of this section, the != operator is again **promoted** from a scalar operator to a component-wise application between the two structured values. … but this will end in an interesting compiler error:

```
error: promoted expression "!=" used in while condition
```

Wasn't that what we wanted? Well, halfway! What we actually expressed with `Vector_1 != Vector_2` is a vector of boolean values – a type which is not fit for a while condition. What still needs to be done is to **reduce** (or **fold**) this boolean vector into a single boolean value. So let's say so:

```
while || reduce (Vector_1 != Vector_2) do {
  ...
}
```

Now we **reduce** the boolean vector by combining all values with a boolean "or" ("||" in Chapel syntax).

Using the mathematical notation of $\vee$ for boolean "or", this boolean expression would translate as a serial implementation into:

$$\vec{v_1} \neq \vec{v_2} \Rightarrow \begin{pmatrix} a_1 \\ a_1 \\ \dots \\ a_n \end{pmatrix} \neq \begin{pmatrix} b_1 \\ b_1 \\ \dots \\ b_n \end{pmatrix} \Rightarrow (a_1 \neq b_1) \vee (a_2 \neq b_2) \vee \dots \vee (a_n \neq b_n)$$

As you saw in the "Reduce" exercise from lab 6: The reduction itself cannot be brought down to $\Theta(1)$ (regardless how many concurrent core you have), but it can potentially be accelerated to $\Theta(\log n)$ (given enough resources), which is still significant.

And you guessed right: your Chapel compiler will automatically apply maximal, physical concurrency on both stages of this comparison.

Not all operations on complex data structures are strictly component-wise. Sometimes we need to add some context to the operations. Do not despair: We might still get away without any explicit concurrency.

You already saw an example of concurrent operations with context at the very beginning of the course:

```
config const n              = 100,
             max_iterations = 50,
             epsilon        = 1.0E-5,
             initial_border = 1.0;

const Matrix_w_Borders = {0 .. n + 1, 0 .. n + 1, 0 .. n + 1},
      Matrix           = Matrix_w_Borders [1 .. n, 1 .. n, 1 .. n],
      Single_Border    = Matrix.exterior (1, 0, 0);

var Field      : [Matrix_w_Borders] real,
    Next_Field : [Matrix]           real;

proc Stencil (M : [/* Matrix_w_Borders */] real,
                                              (i, j, k) : index (Matrix)) : real {

  return (M [i - 1, j, k]
        + M [i + 1, j, k]
        + M [i, j - 1, k]
        + M [i, j + 1, k]
        + M [i, j, k + 1]
        + M [i, j, k - 1]) / 6;
}

Field [Single_Border] = initial_border;

for l in 1 .. max_iterations {

  forall Matrix_Indices in Matrix do
    Next_Field [Matrix_Indices] = Stencil (Field, Matrix_Indices);

  const delta = max reduce abs (Field [Matrix] - Next_Field);

  Field [Matrix] = Next_Field;

  if delta < epsilon then break;
}
```

On the surface the `Stencil` function is applied to every cell in the matrix individually, So why couldn't we use the operator promotion technique from above? If you look closer you will find that the `Stencil` function requires knowledge about the position of the cell, such that it can extract information about the neighbourhood. This context information (namely the cell indices) is provided in the `forall` loop, which works logically like a `for` loop, but provides the additional hint that the compiler is free to parallelise the iterations according to available hardware. The `forall` loop construct allows for more flexibility while still keeping the concurrency forms of the implementation automated.

We could have used those `forall` loops instead of the promoted operations all along:

```
var Addition : [Vector] real;
forall i in Vector do {
  Addition (i) = Vector_1 (i) + Vector_2 (i);
}
```

yet the promoted expression:

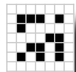
```
const Addition : [Vector] real = Vector_1 + Vector_2;
```

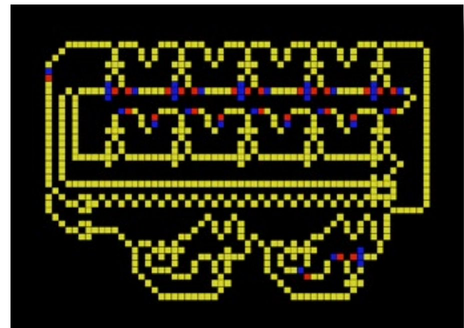
is much shorter, more readable and has exactly the same concurrency effect (in Chapel).

Exercise 1: Wireworld

Wireworld was once the first major assignment in the first computer science course at the ANU. Back then it was a bit of a challenge for many students (maybe for you too?). Either way, this is three semesters later and you are now equipped with many more tools and knowledge to handle such a job. Of course we will handle everything concurrently now.

To refresh the memory of most, and to introduce the concept to our newcomers: a **cellular automaton**, which is itself a version of a **finite state machine** without external inputs during execution. The **automaton state** is a regular (usually finite) grid of **cells**, of which each is in exactly one of a discrete (and usually small) number of **cell states**. The cellular automaton commonly updates its overall state atomically and by applying the same state-transition-rule to each cell simultaneously. The probably best know example of a **cellular automaton** is John Horton Conway's **Game of Life**, which uses a rectangular grid and two states per cell ("Dead" ☐ or "Alive" ■). Cells are updated by counting the number of alive cells around them and transition into a new state based on a simple rule. For background: Many nontrivial cellular automata are *Turing-complete*, i.e. informally: they can run any algorithm which your computer in front of you can run as well.





Wireworld is another version of a cellular automaton. Like Conway's **Game of Life** it uses a rectangular grid of cells, yet there are four possible states per cell:

- ■ *Empty*
- ■ *Electron head*
- ■ *Electron tail*
- ■ *Conductor*

and the accompanying rule to progress each cell into the next state:

- ■ *Empty* ⇒ ■ *Empty*
- ■ *Electron head* ⇒ ■ *Electron tail*
- ■ *Electron tail* ⇒ ■ *Conductor*
- ■ *Conductor* ⇒ { ■ *Electron head* ; if 1 or 2 of the 8 adjacent cells are *Electron heads* / ■ *Conductor* ; otherwise }

This is a sufficient rule to enable any form of logic circuit to be emulated, so you could in principle "draw" you whole computer in it and let it run as a colourful animation. As you can "build" your actual computer with this system, it is obviously also Turing complete.

Cellular automata are begging for concurrent implementations. Back in first semester we didn't have the background to do it – but now we do and plenty of it. Implementing Wireworld concurrently should be a breeze.

As you are not experienced in Chapel so far and I would like you to focus on the concurrent bits instead of looking up syntax rules, I provide the complete data-structures right away:

```
config const n              = 50,
             max_iterations = 1000;

const World_w_Borders = {0 .. n + 1, 0 .. n + 1},
      World           = World_w_Borders [1 .. n, 1 .. n],
      Binary          = {0 .. 1};

type Binary_Range    = index (Binary);

enum Cells {Empty, Conductor, Head, Tail};

proc Match_Count (Cell, Match : Cells) : Binary_Range {

  return (if Cell == Match then 1 else 0);
}

var State      : [World_w_Borders] Cells = Cells.Empty,
    Next_State : [World]           Cells = Cells.Empty;
```

And you already have concurrent code. Can you spot where the above source will result in concurrent code?

This is followed by some rather boring code to initialise the `State` to something (you can check this out in the code which you will download if you like).

Next we define the cell state transition function which is just a few range definitions and otherwise word-by-word copied from the rule above:

```
proc Next_Cell (M : [] Cells, (i, j) : index (World)) : Cells {

  const One_or_Two = {1 .. 2},
        Neighbours = {0 .. 9};

  type Neighbour_Range = index (Neighbours);

  select M [i, j] {
    when Cells.Head      do return Cells.Tail;
    when Cells.Tail      do return Cells.Conductor;
    when Cells.Conductor do {
      const Heads : Neighbour_Range = ??? ;
      return (if One_or_Two.member (Heads) then Cells.Head else Cells.Conductor);
    }
  }
  return Cells.Empty;
}
```

One expression (???) is missing and you will need to fill this one in. You need to calculate the expression concurrently and in a single line. With the knowledge which you accumulated from the interlude, this is still not trivial, but ultimately obvious once you got it.

Then the last thing to do is to make the whole world go. This is even simpler than the cell transition:

```
for l in 1 .. max_iterations {

  ??? // transition the current State to the Next_State

  if ??? then {
    writeln (“World static at generation: “, l);
    break;
  }
  else {
    State [World] = Next_State;
  }
}
```

To keep it interesting there are two small bits missing. The first one is the transition of the whole world. You of course need to implement this concurrently and you are only allowed to use two lines for it. The missing boolean expression after the `if` needs to express that the current and the next state are identical (obviously you want this expression to be evaluated concurrently as well). No more than one line allowed for this one.

And finally: take a guess whether the last assignment in the given code could be run concurrently as well?

Use any text editor you like for your Chapel programming. Syntax highlighting and basic indention handling seems to work in TextMate, vim and emacs, but there will be others. Check what's available for your favourite editor and let us know if you find something nice.

Compiling your program is simply done on the command line with:

```
chpl -o Wireworld Wireworld.chpl
```

The course-site has the link to download the compiler from Cray (already installed in the labs for you). If all goes through without issues then there is no message besides that you will find an executable file `Wireworld` in the same directory.

Once you got your program past the Chapel compiler, review the whole program again and take mental notes of all the parts which will run concurrently without you ever having said "`task`".

Submit your program by direct copy-and-paste to the *SubmissionApp* under "Lab 8 Wireworld" for code review by us.

Interlude: Implicit concurrency for the win?

While you just saw extremely short programs with maximal concurrency impact, you may wonder why we didn't show you all this in the first place and left those synchronization issues for other people to solve (for example those writing the Chapel compiler for you)?

Unfortunately not all applications which you might need to address are neatly following the patterns of vector processing, maps and folds. Pragmatic languages like Chapel reflect this by the fact that they also have the means to express explicit concurrency. We will look into those options soon and you will be glad to have your concurrent programming concepts well prepared from your previous six labs in Ada before going there.

So: all of this (implicit or explicit concurrency) will be essential and valuable in practice. As always: it is all about the right tool for the job – not about finding the one magic key.

Exercise 2: Game of Life

Re-write the program which you gained in exercise 1 to express Conway's **Game of Life** (you can also start from a minimal code framework for this from the course-site or take it on on your own). Add the means to detect that the world finds itself in a loop (a previous state reappears), so that you can break out of the simulation if that happens. For the sake of finite memory you may want to constraint your loop detection to a certain lengths.

As the cell states for this automaton are only two: Could you think of a more efficient representation of those states? Would this enhance performance?

Submit your program by direct copy-and-paste to the *SubmissionApp* under "Lab 8 Game of Life" for code review by your peers and us.