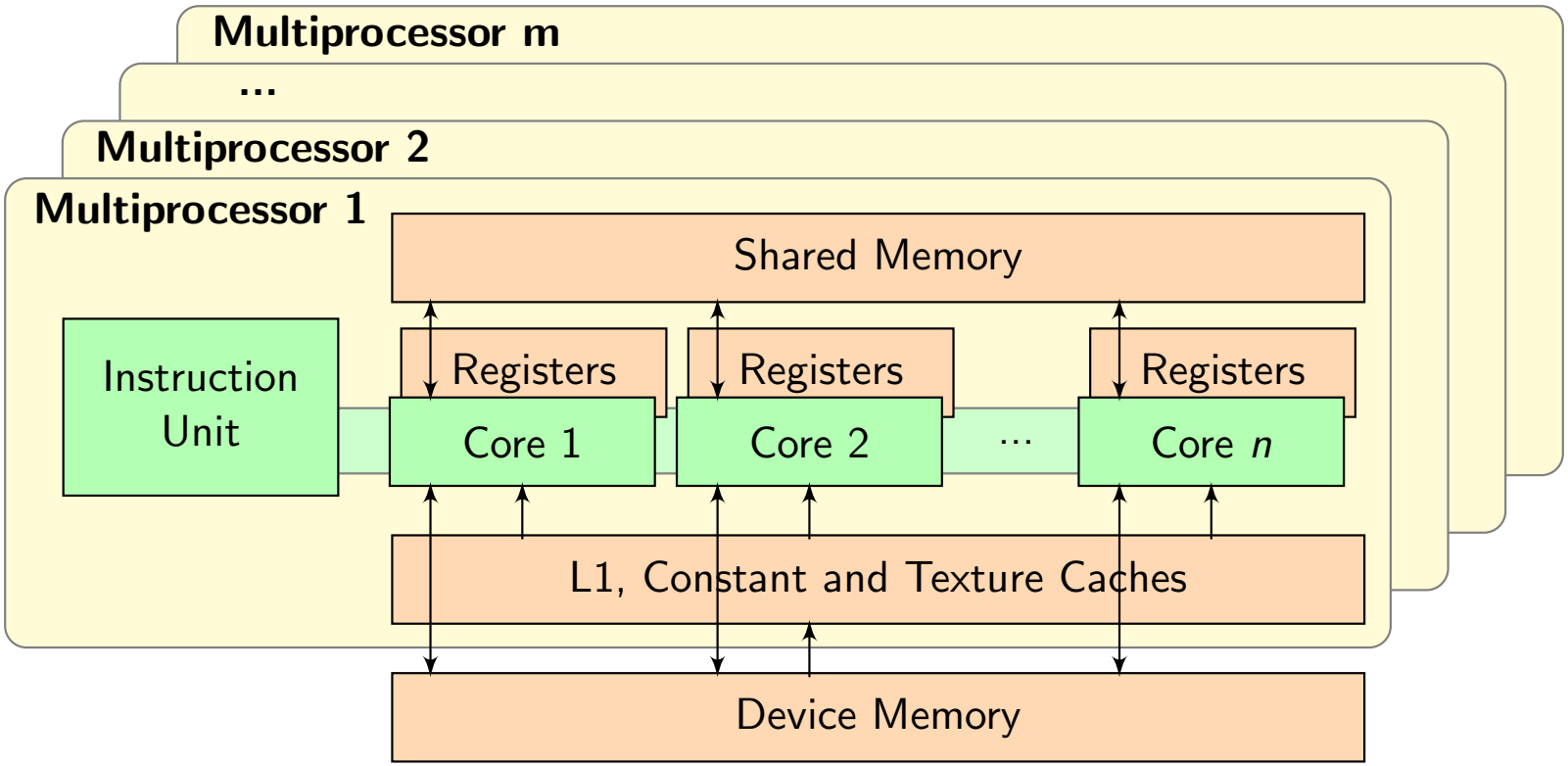
Synthesising Graphics Card Programs from DSLs

Luke Cartey[1] Oege de Moor[1] Rune Lyngsoe[2]

[1]Department of Computer Science, [2]Department of Statistics
University of Oxford

11th June 2012

GPU development is difficult



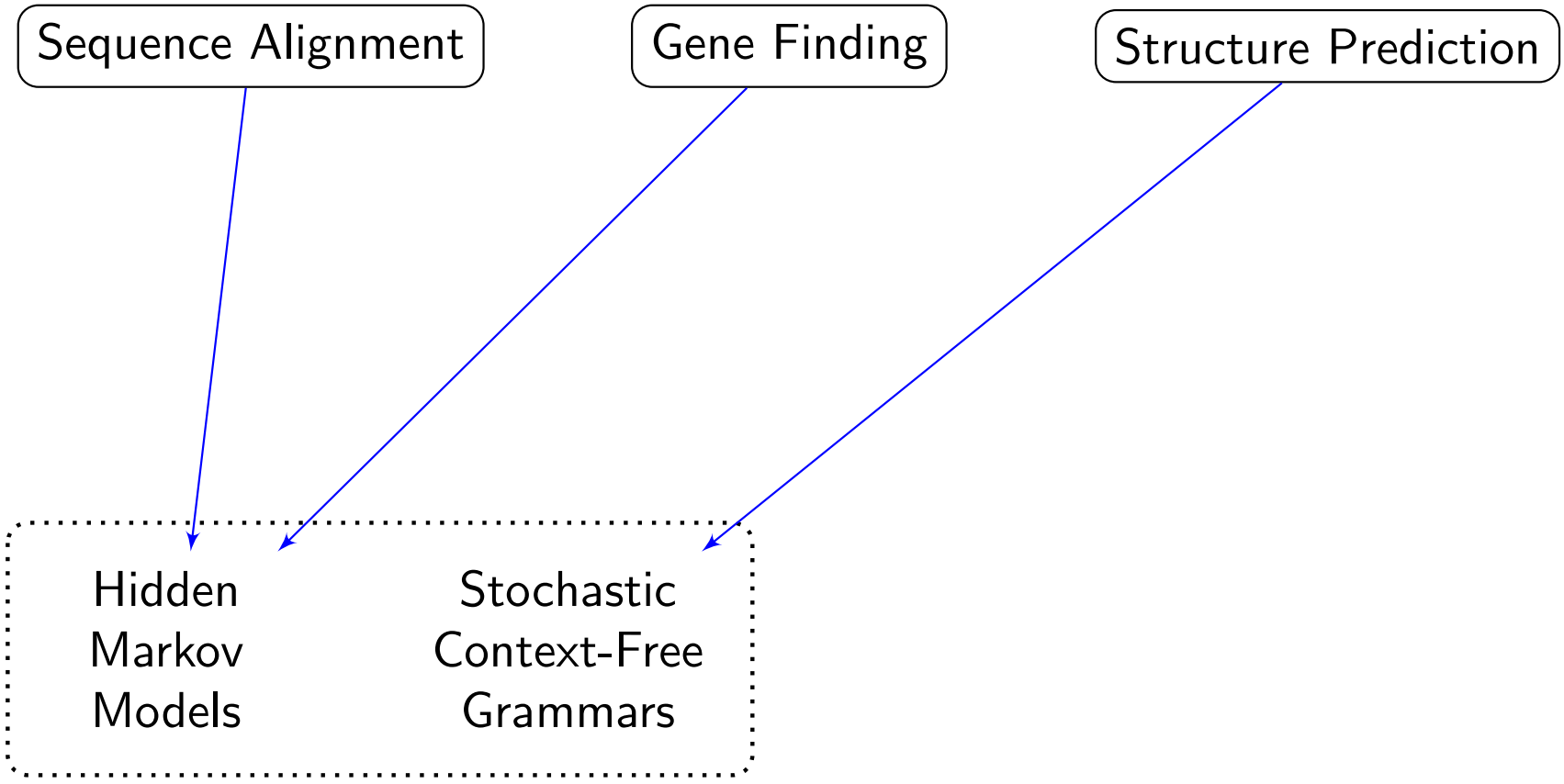
Case Study in Bioinformatics

Sequence Alignment

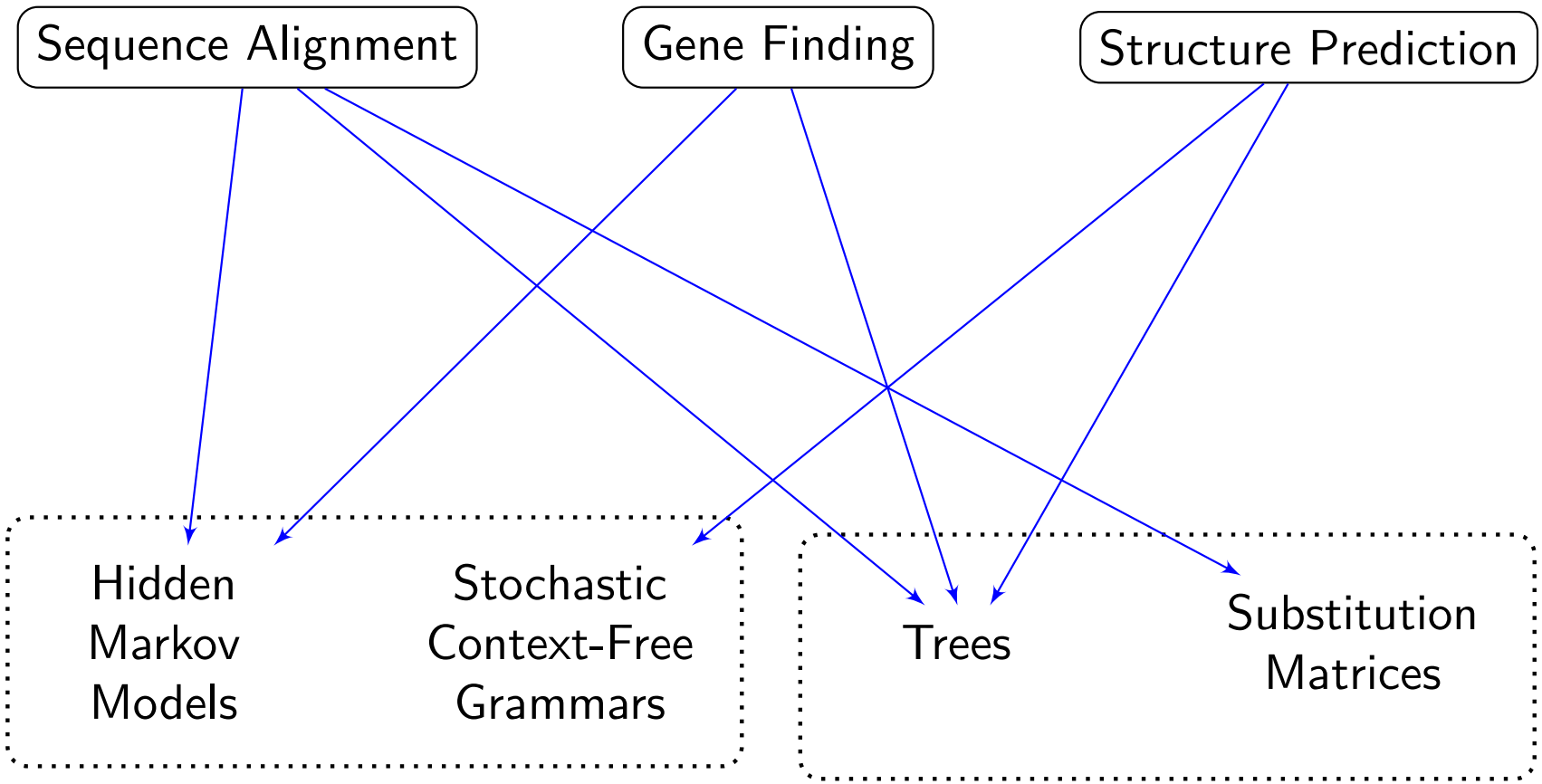
Gene Finding

Structure Prediction

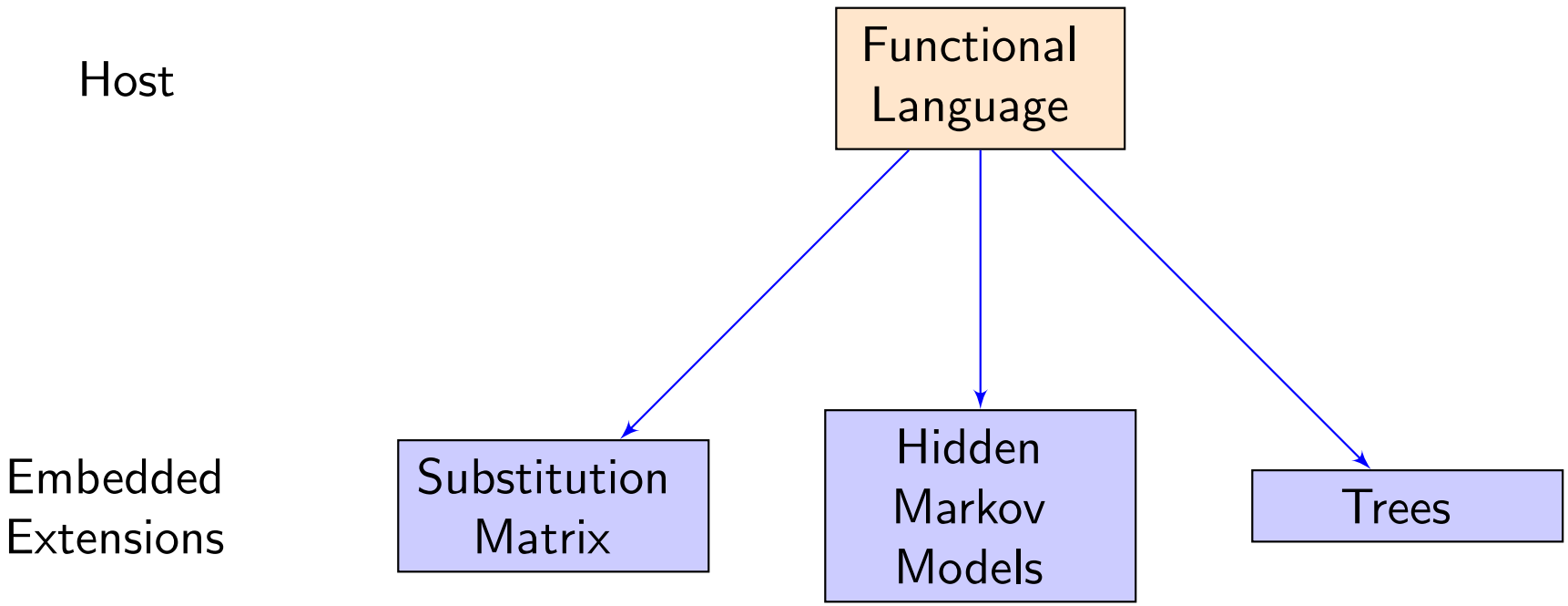
Case Study in Bioinformatics



Case Study in Bioinformatics



Bioinformatics DSL Framework



Language example - edit distance

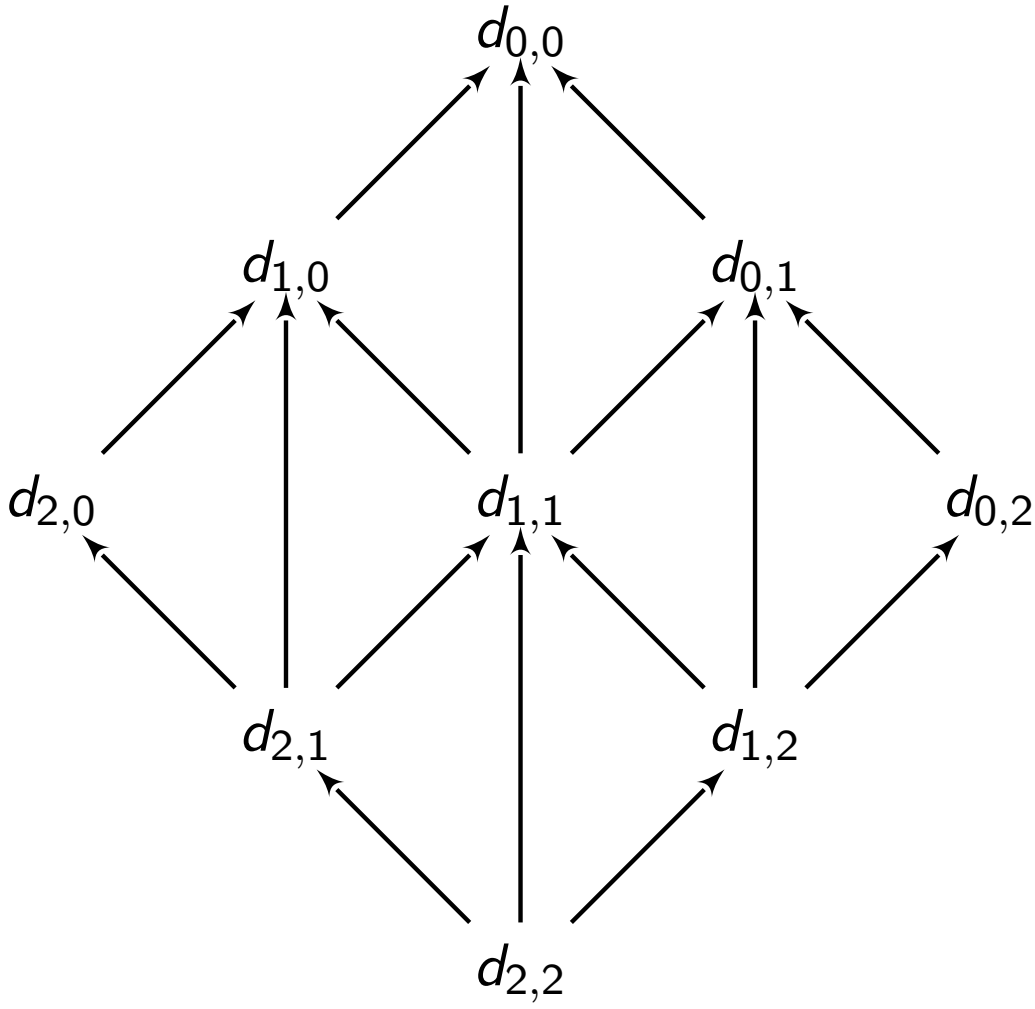
$$d(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ d(i-1,j-1) & \text{if } s[i] = t[j] \\ min \begin{pmatrix} d(i-1,j), \\ d(i,j-1), \\ d(i-1,j-1) \end{pmatrix} + 1 & \text{otherwise} \end{cases}$$

Language example - edit distance

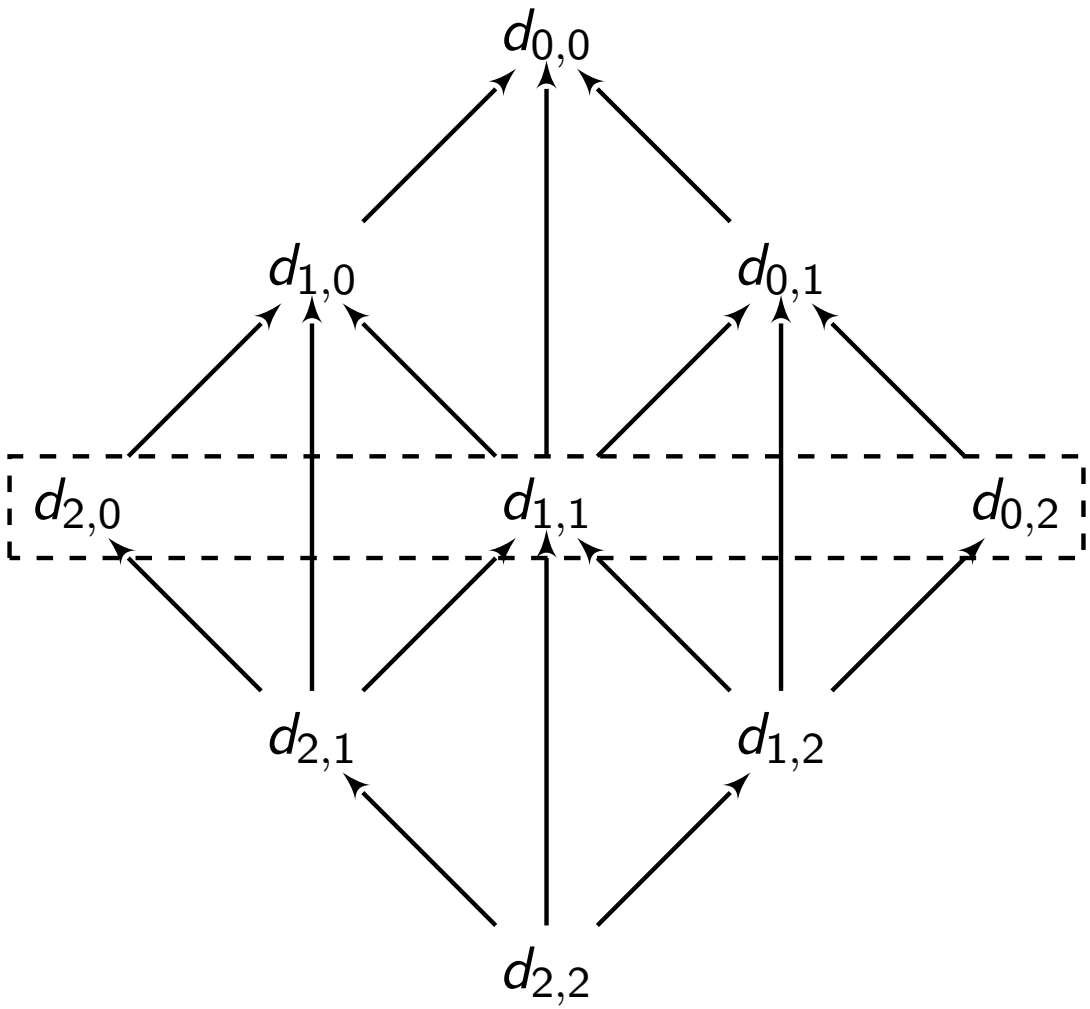
$$d(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ d(i-1,j-1) & \text{if } s[i] = t[j] \\ min \begin{pmatrix} d(i-1,j), \\ d(i,j-1), \\ d(i-1,j-1) \end{pmatrix} + 1 & \text{otherwise} \end{cases}$$

```
int d(seq[en] s, index[s] i, seq[en] t, index[t] j) =
   if i == 0 then
      j
   else if j == 0 then
      i
   else if s[i - 1] == t[j - 1] then
      d(i - 1, j - 1)
   else
      (d(i - 1, j) min d(i, j - 1) min d(i - 1, j - 1)) + 1
```

Dependency Graph



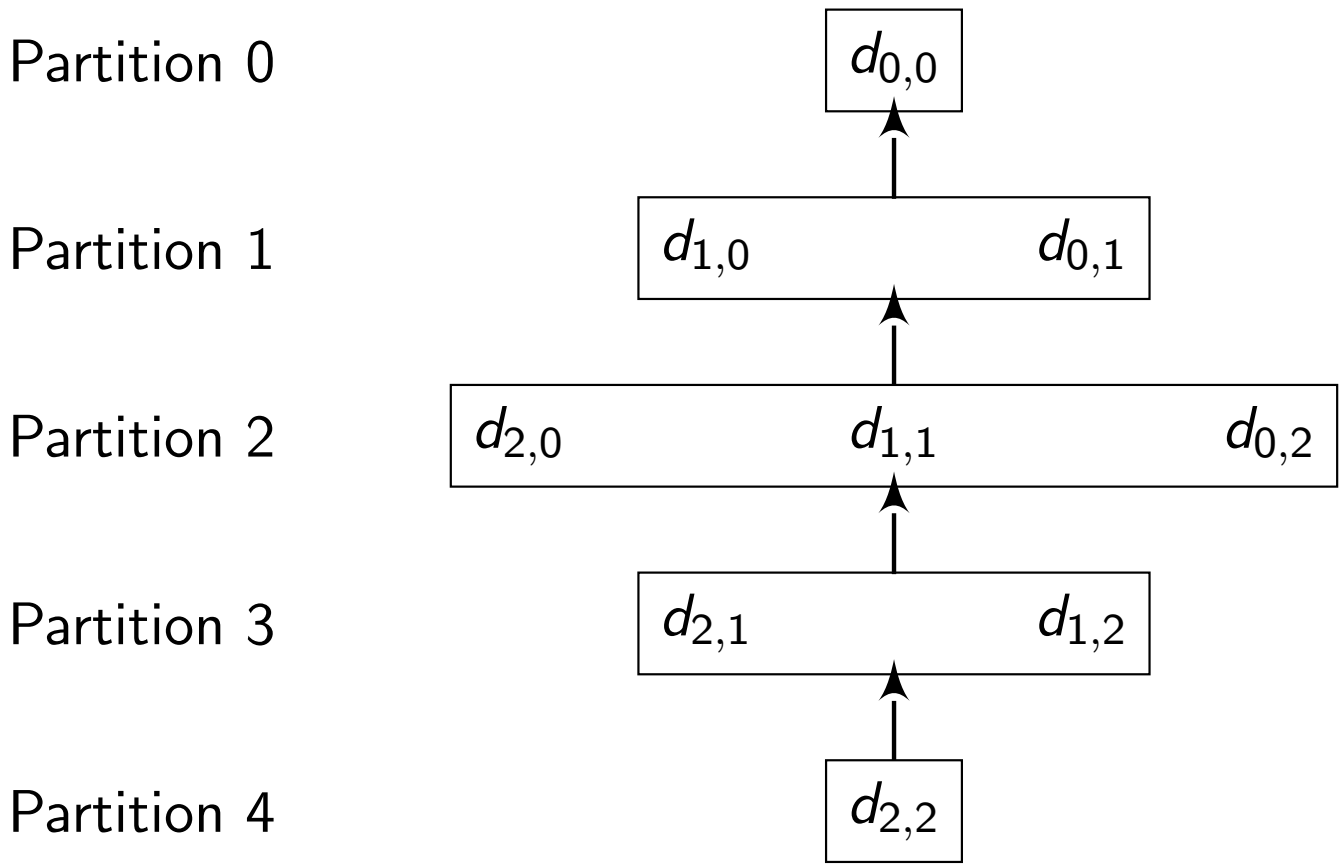
Dependency Graph



Schedule

Linear function that describes partitions of **independent** values.

e.g $S_d(i, j) = i + j$



Valid Schedules

Partition Ordering

$$\forall (i_1, j_1) \in P, (i_2, j_2) \in Q : (i_1, j_1) \rightsquigarrow (i_2, j_2) \implies P > Q$$

Where $\rightsquigarrow$ is the transitive closure of dependencies ($\rightarrow$)

Valid Schedules

Partition Ordering

$$\forall (i_1, j_1) \in P, (i_2, j_2) \in Q : (i_1, j_1) \rightsquigarrow (i_2, j_2) \implies P > Q$$

Where $\rightsquigarrow$ is the transitive closure of dependencies ($\rightarrow$)

$$d(i, j) = \ldots d(i-1, j) \ldots$$

$$\{(i, j) \rightarrow (i-1, j) | \forall i, j\}$$

Valid Schedules

Partition Ordering

$$\forall (i_1, j_1) \in P, (i_2, j_2) \in Q : (i_1, j_1) \rightsquigarrow (i_2, j_2) \implies P > Q$$

Where $\rightsquigarrow$ is the transitive closure of dependencies ($\rightarrow$)

$$d(i, j) = \ldots d(i-1, j) \ldots$$

$$\{(i, j) \rightarrow (i-1, j) | \forall i, j\}$$

Valid schedules adhere to: $\forall i, j, d(i_r, j_r) : S_d(i, j) > S_d(i_r, j_r)$

Identify a schedule

Problem: Find a $S_d(i,j) = a_1 i + a_2 j$.

Step 1: Identify criteria on S_d

For each $d(i,j) = \ldots d(i_r, j_r) \ldots$ use $S_d(i,j) > S_d(i_r, j_r)$

Example:

$$S_d(i,j) > S_d(i-1,j)$$

$$\equiv$$

$$a_1 i + a_2 j - (a_1(i-1) + a_2 j) > 0$$

$$\equiv$$

$$a_1 > 0$$

Identify a schedule 2

Step 2: Construct a CSP to choose a *valid* and *efficient* schedule

Identify a schedule 2

Step 2: Construct a CSP to choose a *valid* and *efficient* schedule

Efficiency criteria: **Minimise** the number of partitions required to evaluate the entire table:

$$min_{a_1,a_2}(max_{x,y}(a_1x + a_2y) - min_{x,y}(a_1x + a_2y))$$

Identify a schedule 2

Step 2: Construct a CSP to choose a *valid* and *efficient* schedule

Efficiency criteria: **Minimise** the number of partitions required to evaluate the entire table:

$$min_{a_1,a_2}(max_{x,y}(a_1x + a_2y) - min_{x,y}(a_1x + a_2y))$$

Solve using up to 2^n CSP problems, where:
$a_i < 0 \implies x_i = 0$
$a_i \geq 0 \implies x_i = xn_i$

Identify a schedule 2

Step 2: Construct a CSP to choose a *valid* and *efficient* schedule

Efficiency criteria: **Minimise** the number of partitions required to evaluate the entire table:

$$min_{a_1,a_2}(max_{x,y}(a_1x + a_2y) - min_{x,y}(a_1x + a_2y))$$

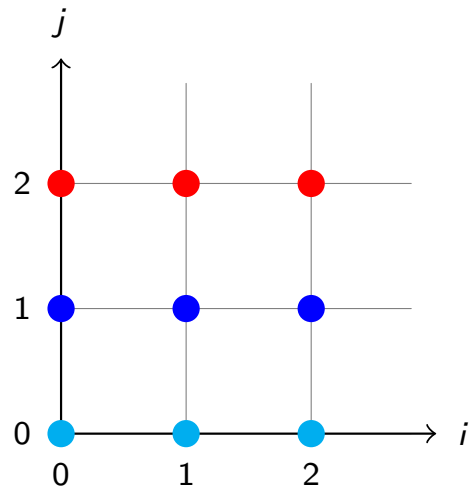
Solve using up to 2^n CSP problems, where:
$a_i < 0 \implies x_i = 0$
$a_i \geq 0 \implies x_i = xn_i$

Finds $S_d(i, j) = i + j$ in edit distance example.

Polyhedral model

Given a schedule + original function, we can generate code using the **polyhedral model**.

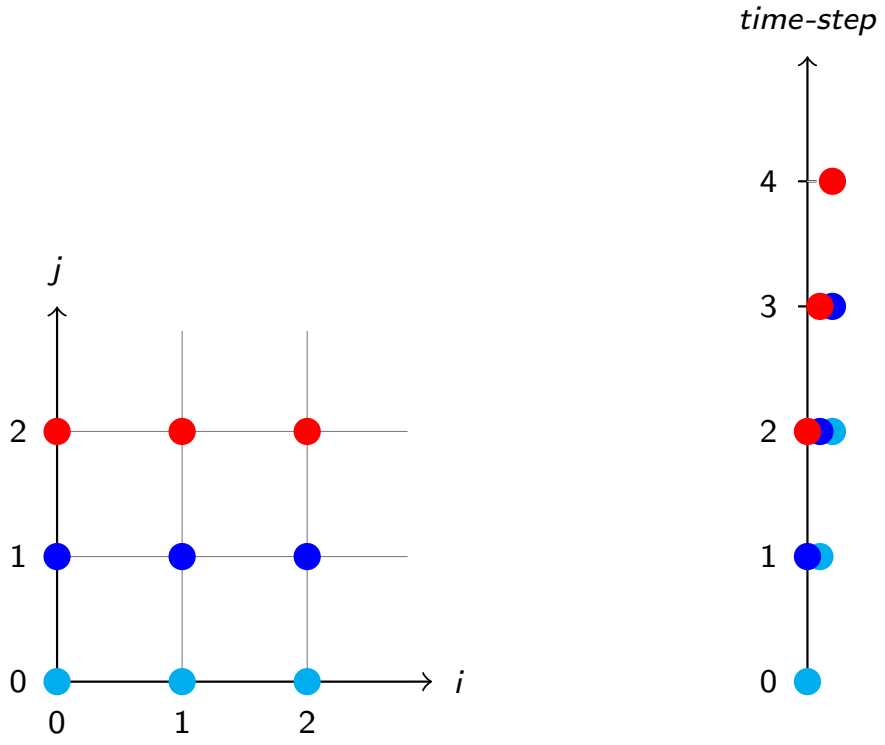
Domain of recursion: $\{0 \leq i \leq n, 0 \leq j \leq m\}$
Schedule: $S_d(i,j) = i + j$



Polyhedral model

Given a schedule + original function, we can generate code using the **polyhedral model**.

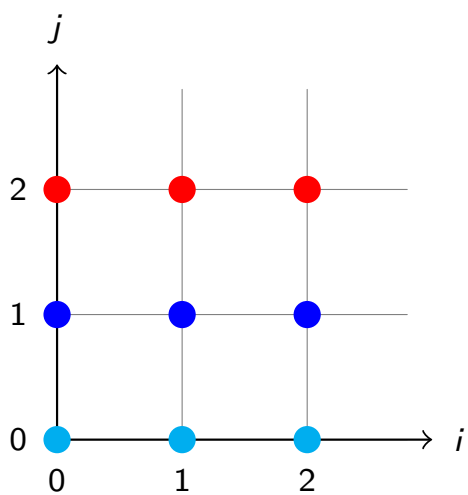
Domain of recursion: $\{0 \leq i \leq n, 0 \leq j \leq m\}$
Schedule: $S_d(i,j) = i + j$

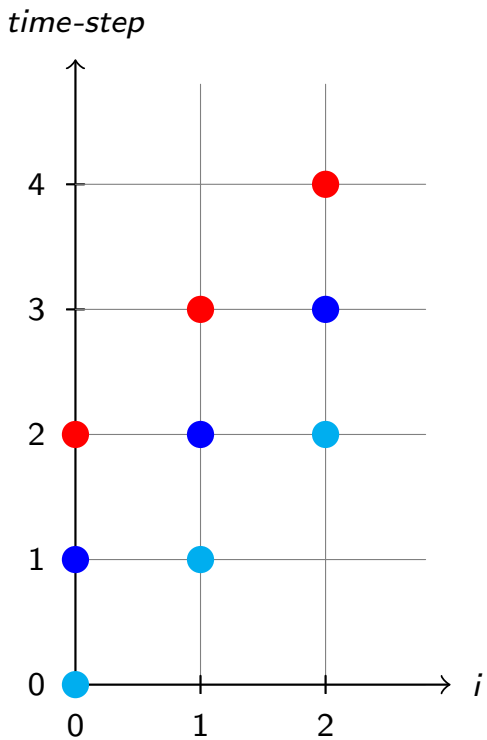


Polyhedral model

Given a schedule + original function, we can generate code using the **polyhedral model**.

Domain of recursion: $\{0 \leq i \leq n, 0 \leq j \leq m\}$
Schedule: $S_d(i,j) = i + j$





Edit Distance Program Synthesis

We use the polyhedral code generator CLooG to produce a set of nested loops that iterate over the transformed domain.

```
for (p=0;p<=m+n;p++) {
  for (i=max(0,p-m);i<=min(n,p);i++) {
    S1(i,p-i);
  }
}
```

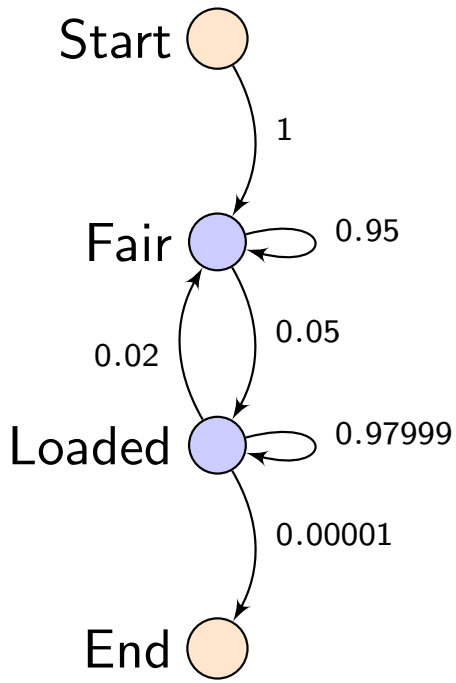
C. Bastoul: Code Generation in the Polyhedral Model is Easier Than You Think. PACT 2004

GPU Program Synthesis

Block size: *tn* threads

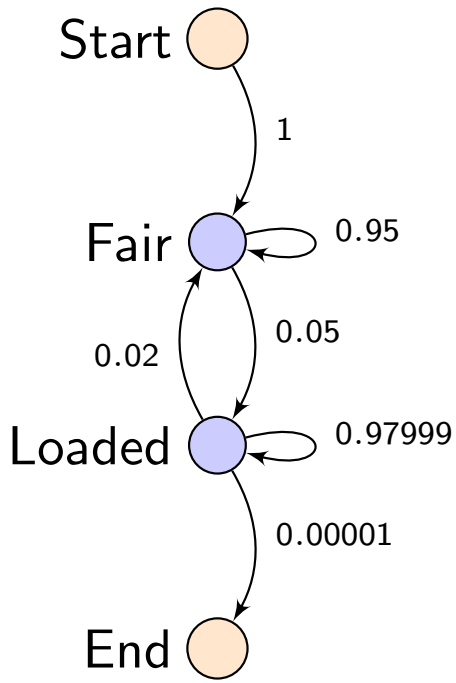
```
parfor threads t in 0..tn {
  for (p=0;p<=m+n;p++) {
    for (i=t+max(0,p-m);i<=min(n,p);i+=tn) {
      j = p - i;
      darr[i,j] = d(i,j);
    }
    sync;
  }
}
```

DSL Extensions - Hidden Markov Models



$$F(q,i) = \sum_{p:a_{p,q}>0} a_{p,q} e_{q,s[i]} F(p, i-1)$$

DSL Extensions - Hidden Markov Models



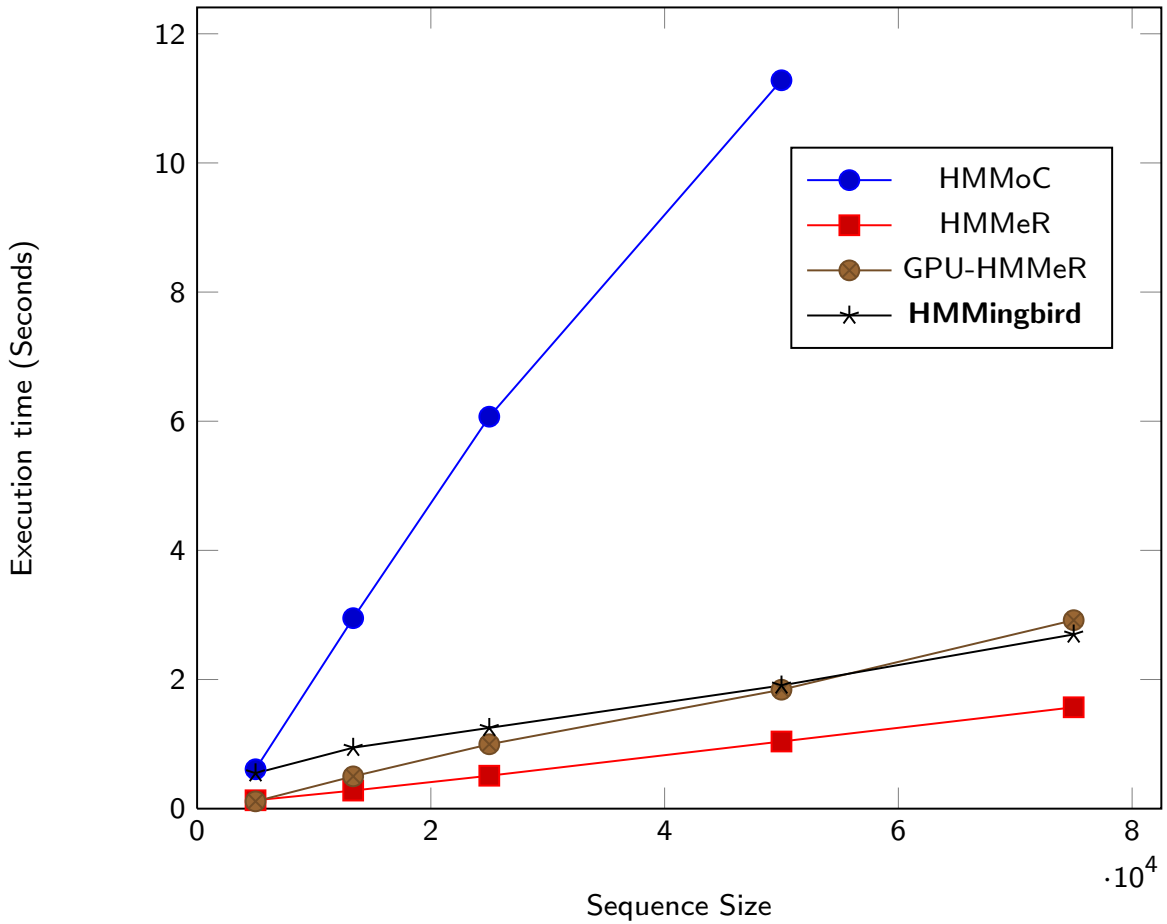
$$F(q,i) = \sum_{p:a_{p,q}>0} a_{p,q} e_{q,s[i]} F(p, i-1)$$

```
define alphabet dice [1,2,3,4,5,6]

define hmm casino {
   alphabet dice;
   startstate start;
   state fair emits fairemission;
   state loaded emits loadedemission;
   endstate end;
   start -> fair 1;
   fair -> fair 0.94999;
   fair -> loaded 0.05;
   fair -> end 0.00001;
   loaded -> loaded 0.97999;
   loaded -> fair 0.02;
   loaded -> end 0.00001;
   emission fairemission =
      [1/6, 1/6, 1/6, 1/6, 1/6, 1/6];
   emission loadedemission =
      [0.1, 0.1, 0.1, 0.1, 0.1, 0.5];
}

prob forward(hmm h, state[h] s, seq[*] x, index[x] i) =
   if i == 0 then
      if s.isstart then 1.0 else 0.0
   else
      sum(t in s.transitionsto :
            forward(t.start, i - 1) * t.prob)
      * (if s.isend then 1.0 else s.emission{x[i-1]})
```

Performance results



Performance for the forward algorithm on a profile HMM model of 10 positions, with varying numbers of sequences.

Conclusion

- We can have our cake and eat it too.
- Automatic parallelisation of first-order, call-by-value functional language.
- High-level approach can provide both an efficient implementation and happy users.