Taming graphics cards for bioinformatics

Luke Cartey

Department of Computer Science
University of Oxford

13th September 2012

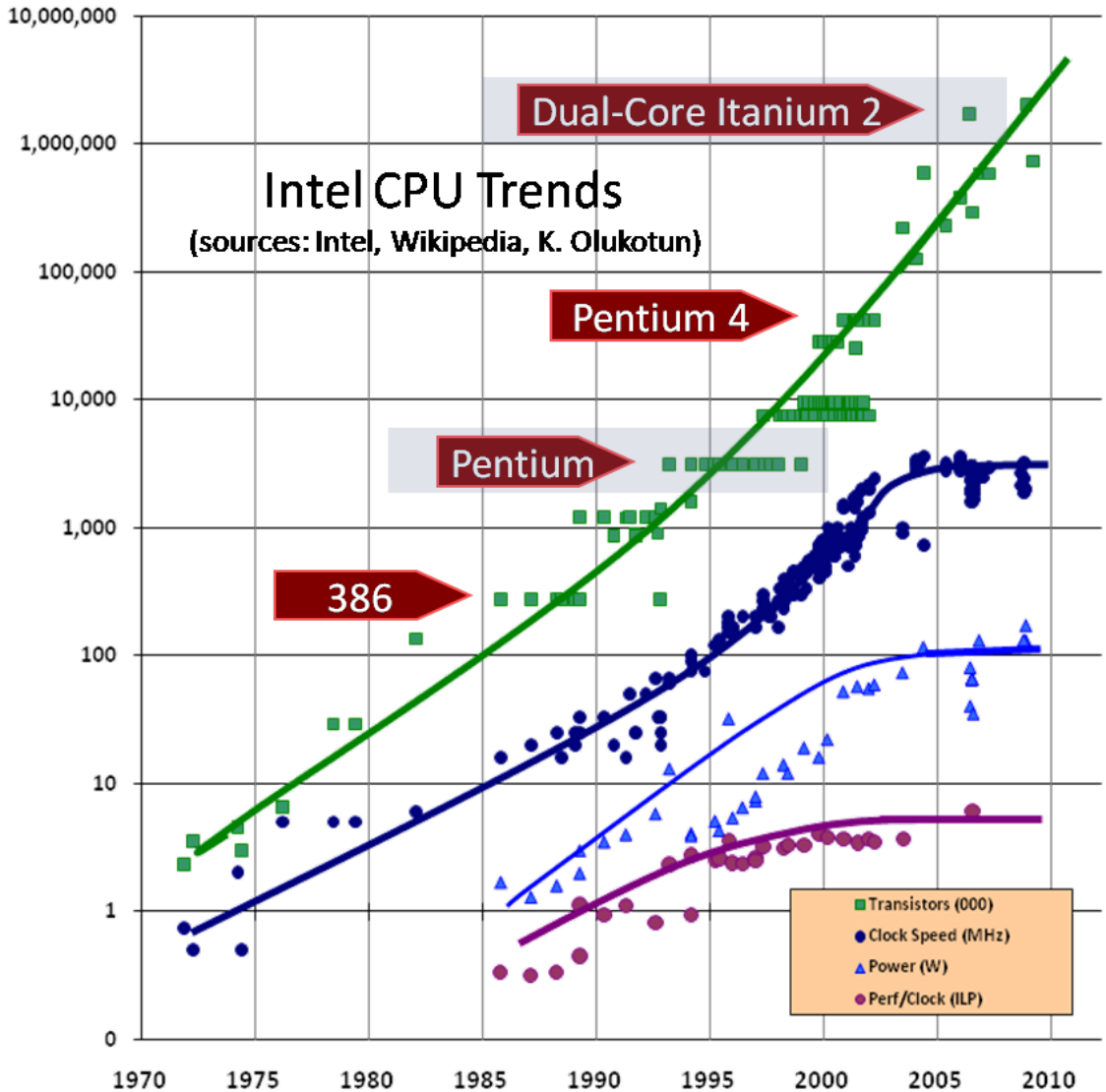
Problems are growing in scope and ambition.

Brings some challenges:

1. We need to make use of larger, more complicated systems in order to run our ambitious programs.
2. How to organise and develop these increasingly complex, long running, applications so that they run correctly and are efficient to write.

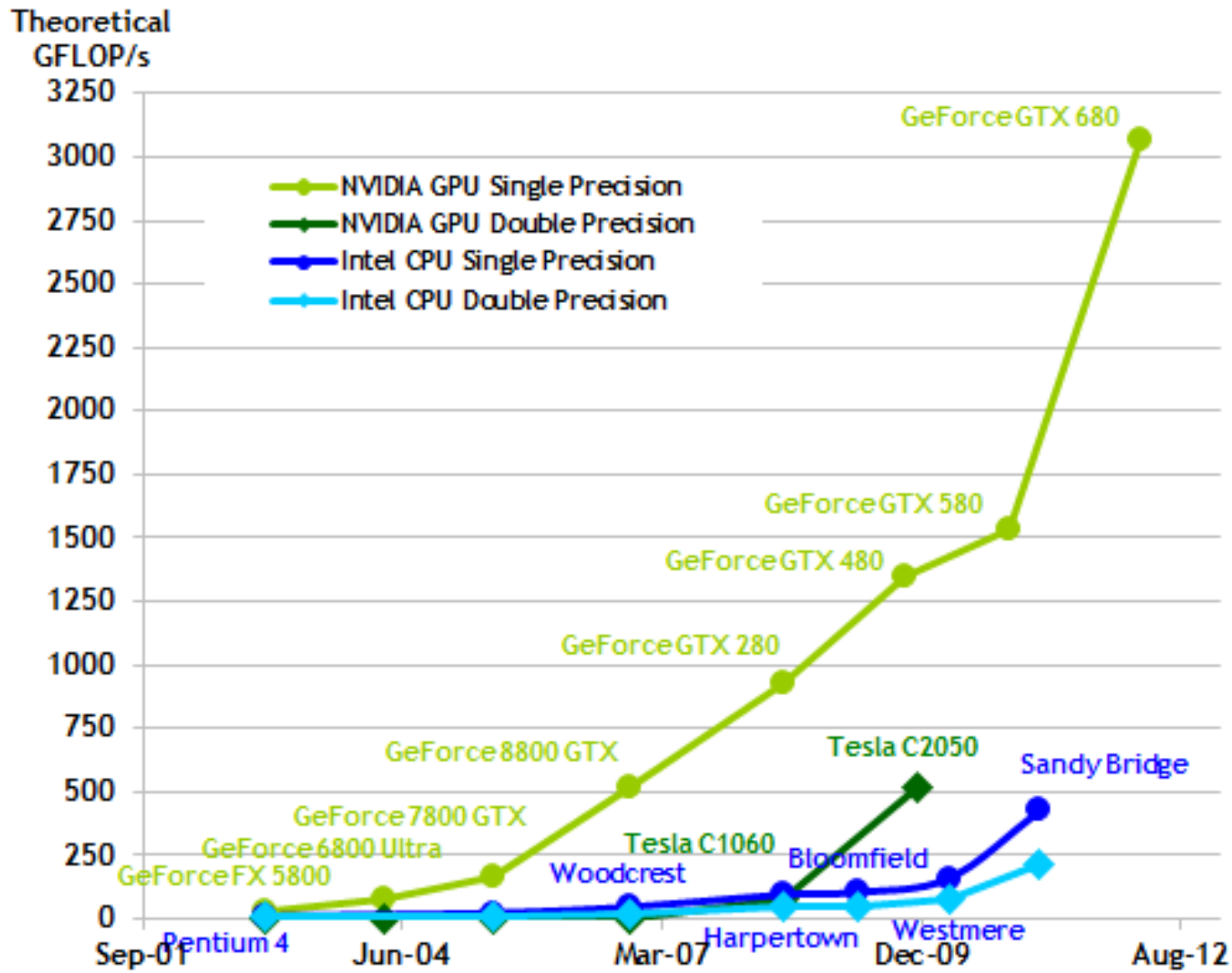
Moore's Law isn't what it was...

...or how I learned to stop worrying and love parallel processing.



Meanwhile...

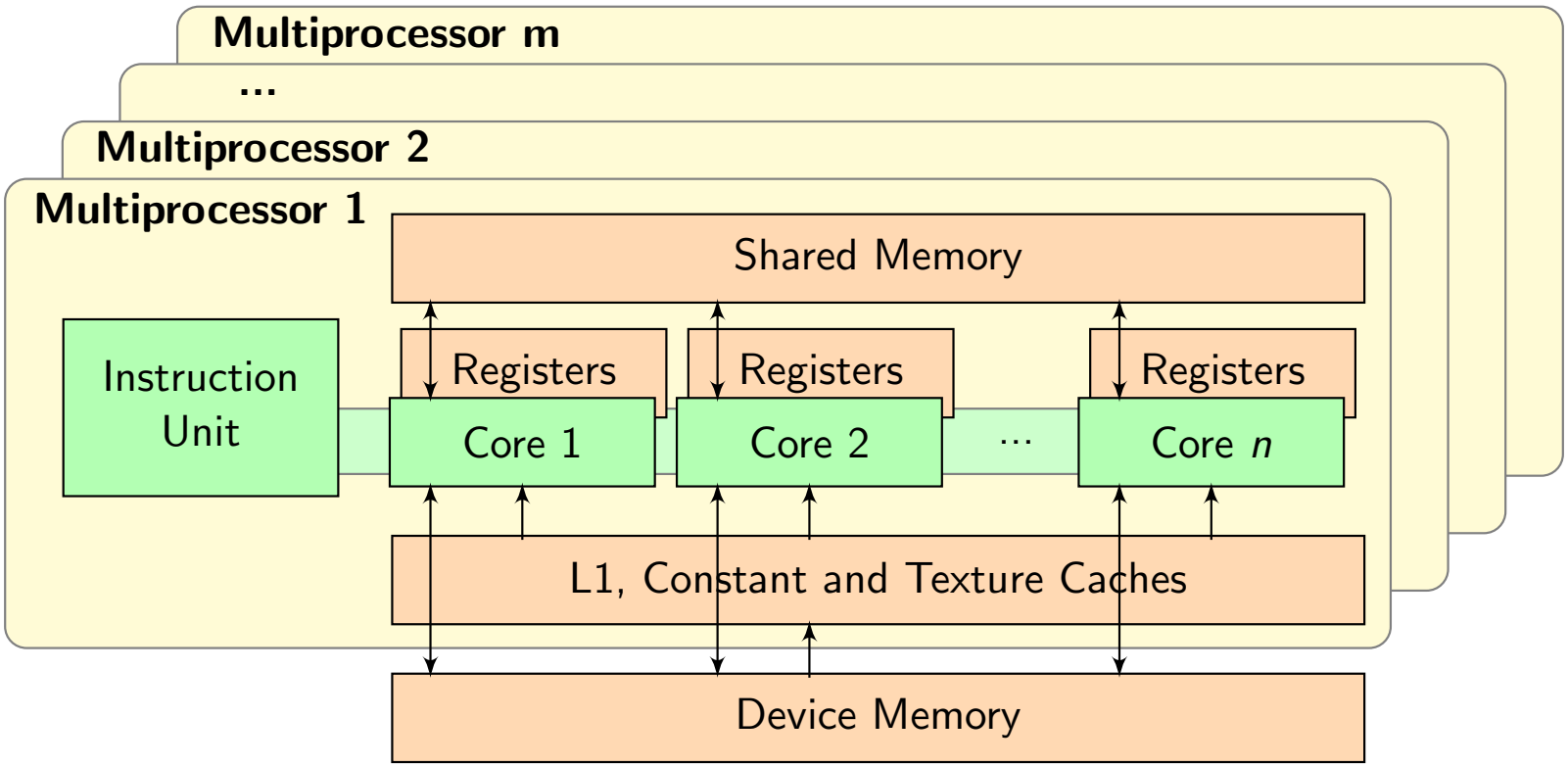
Hardware choices



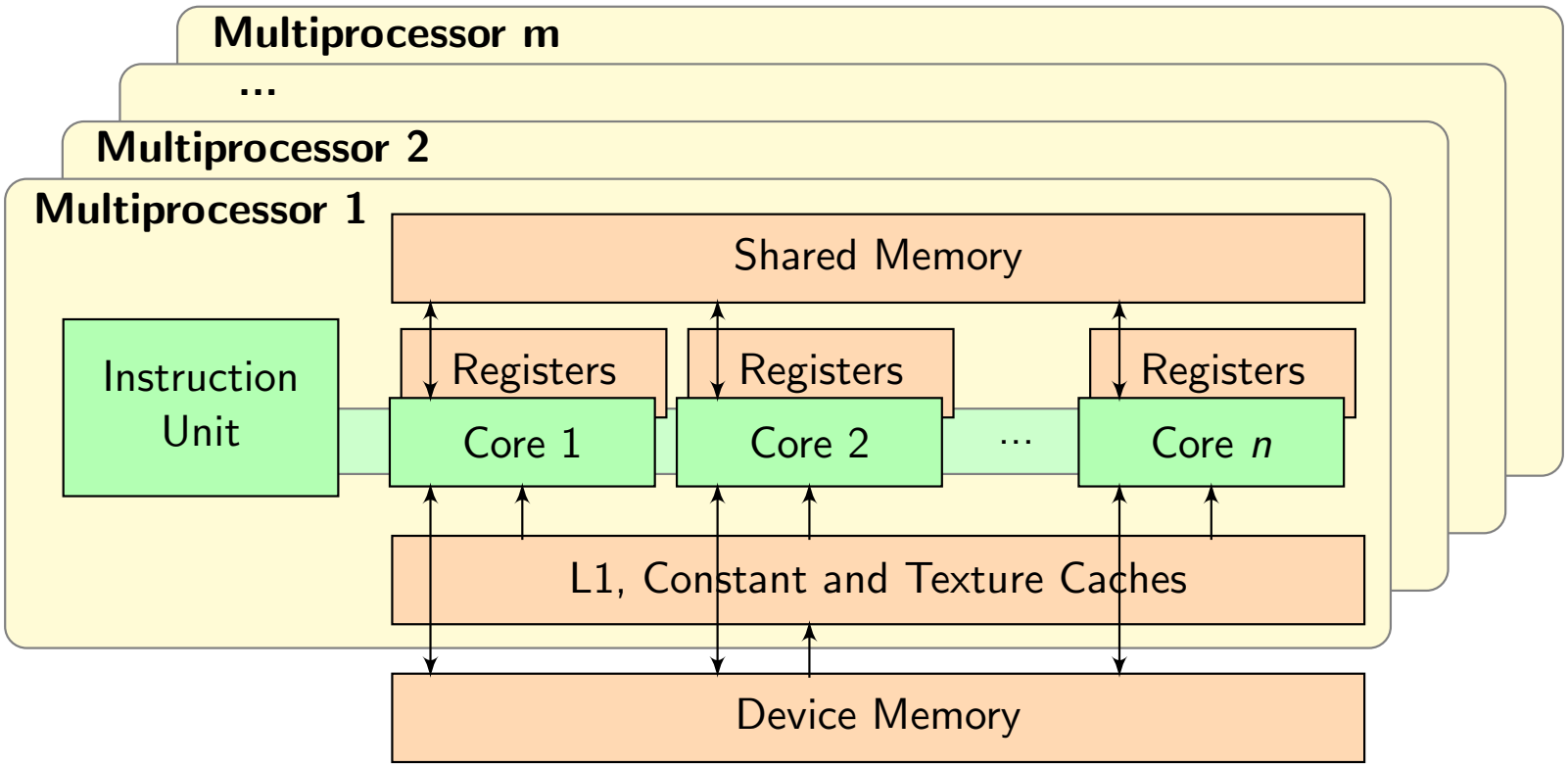
Why GPUs?

- Roughly 20x higher GFLOP/s than comparable CPU.
- Ubiquitous - 300 Million CUDA devices.
- Scales well.

GPU Architecture



GPU Architecture



Tough to use in practice!

We can make graphics cards much easier to use by building "Domain Specific Languages".

Domain Specific Languages

A language that is designed to describe and solve problems in a particular domain.

Domain Specific Languages

A language that is designed to describe and solve problems in a particular domain.

- HTML

```
<h1>A header</h1>
<p>A paragraph <img src="picture.jpg"/></p>
```

- Regexes

```
([a-zA-Z]*)
```

- Latex

```
\section{My section}
This is a \textbf{formula}: $f(x,y) = x + y$.
```

- Make (Ant, etc.)

Domain Specific Languages

A language that is designed to describe and solve problems in a particular domain.

- HTML

```
<h1>A header</h1>
<p>A paragraph <img src="picture.jpg"/></p>
```

- Regexes

```
([a-zA-Z]*)
```

- Latex

```
\section{My section}
This is a \textbf{formula}: $f(x,y) = x + y$.
```

- Make (Ant, etc.)

Describe problems using the "language" of the domain.

Domain Specific Languages

A language that is designed to describe and solve problems in a particular domain.

- HTML

```
<h1>A header</h1>
<p>A paragraph <img src="picture.jpg"/></p>
```

- Regexes

```
([a-zA-Z]*)
```

- Latex

```
\section{My section}
This is a \textbf{formula}: $f(x,y) = x + y$.
```

- Make (Ant, etc.)

Describe problems using the "language" of the domain. Generate code in the language of the target.

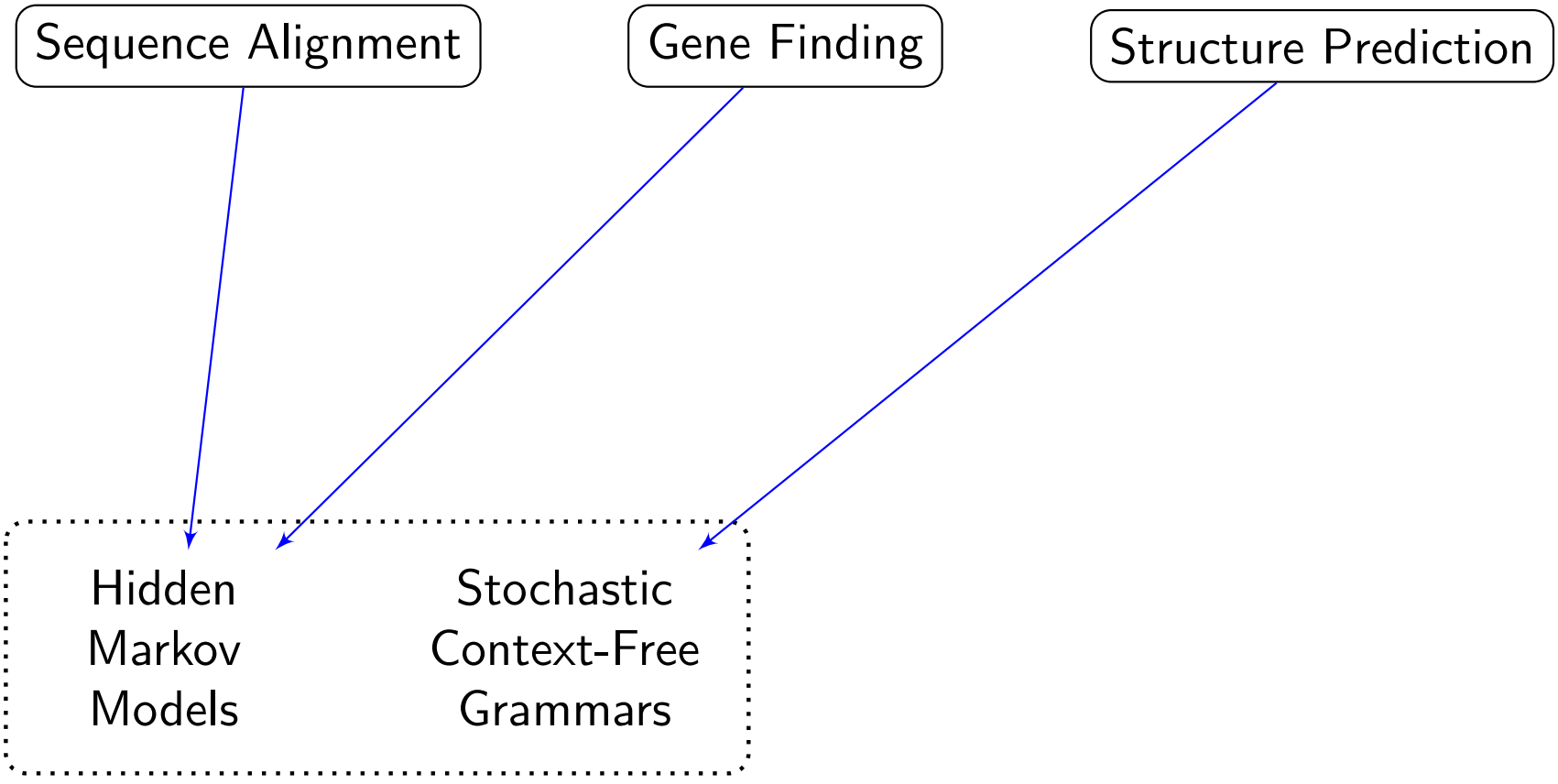
Applications of Interest

Sequence Alignment

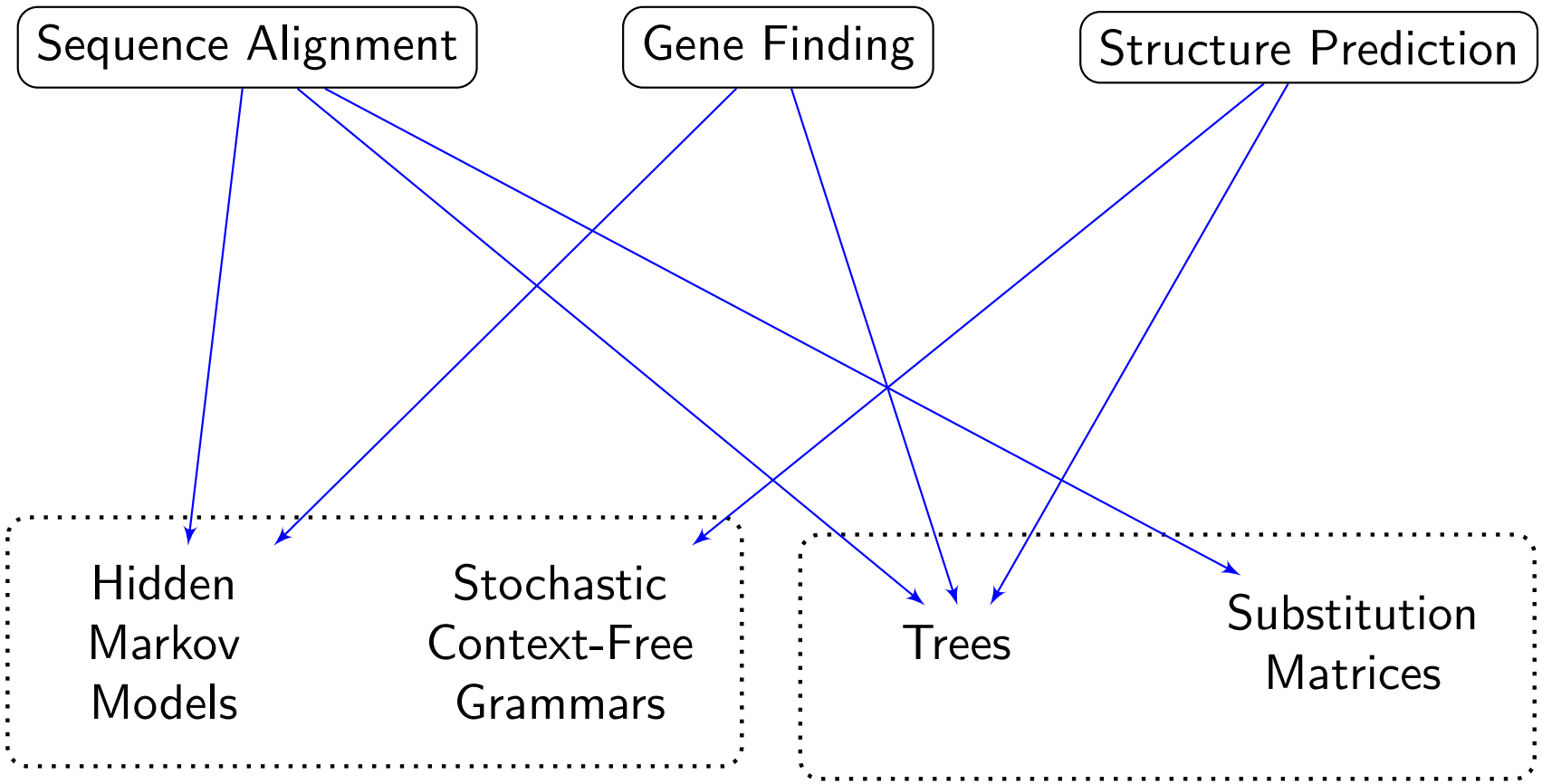
Gene Finding

Structure Prediction

Applications of Interest



Applications of Interest



A common language?

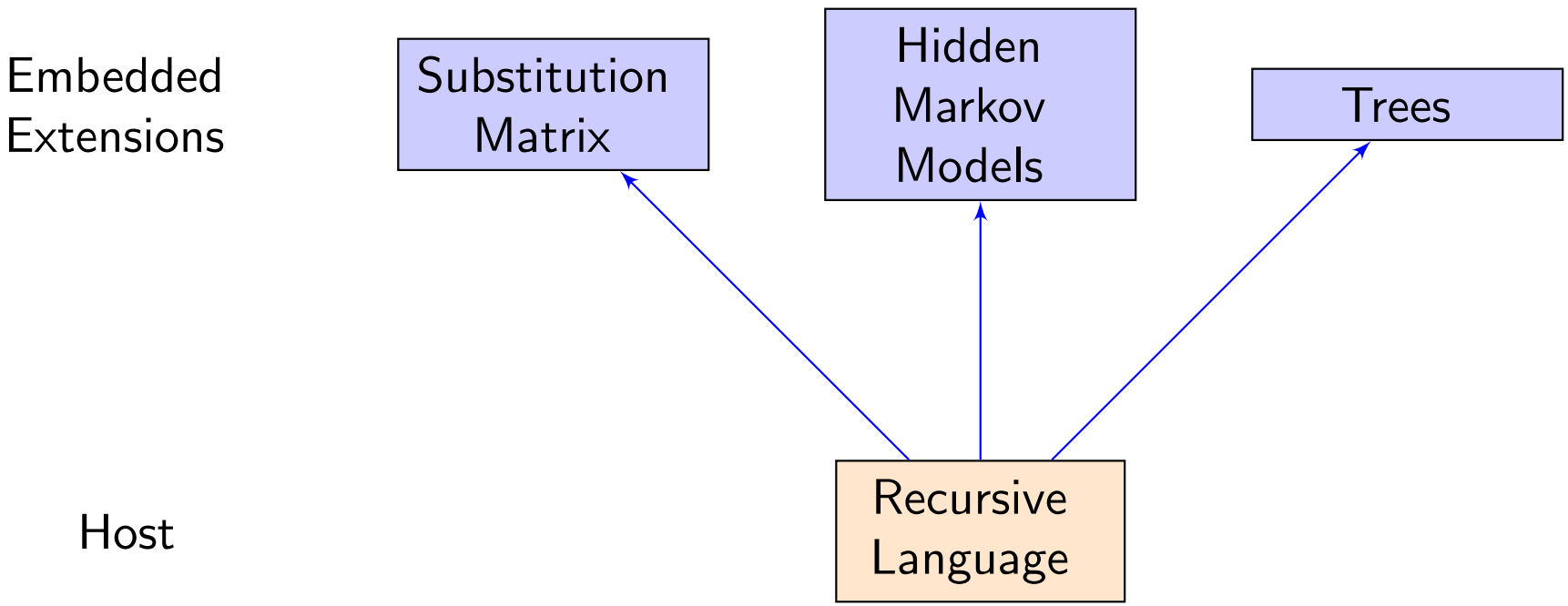
Languages

Substitution Matrix

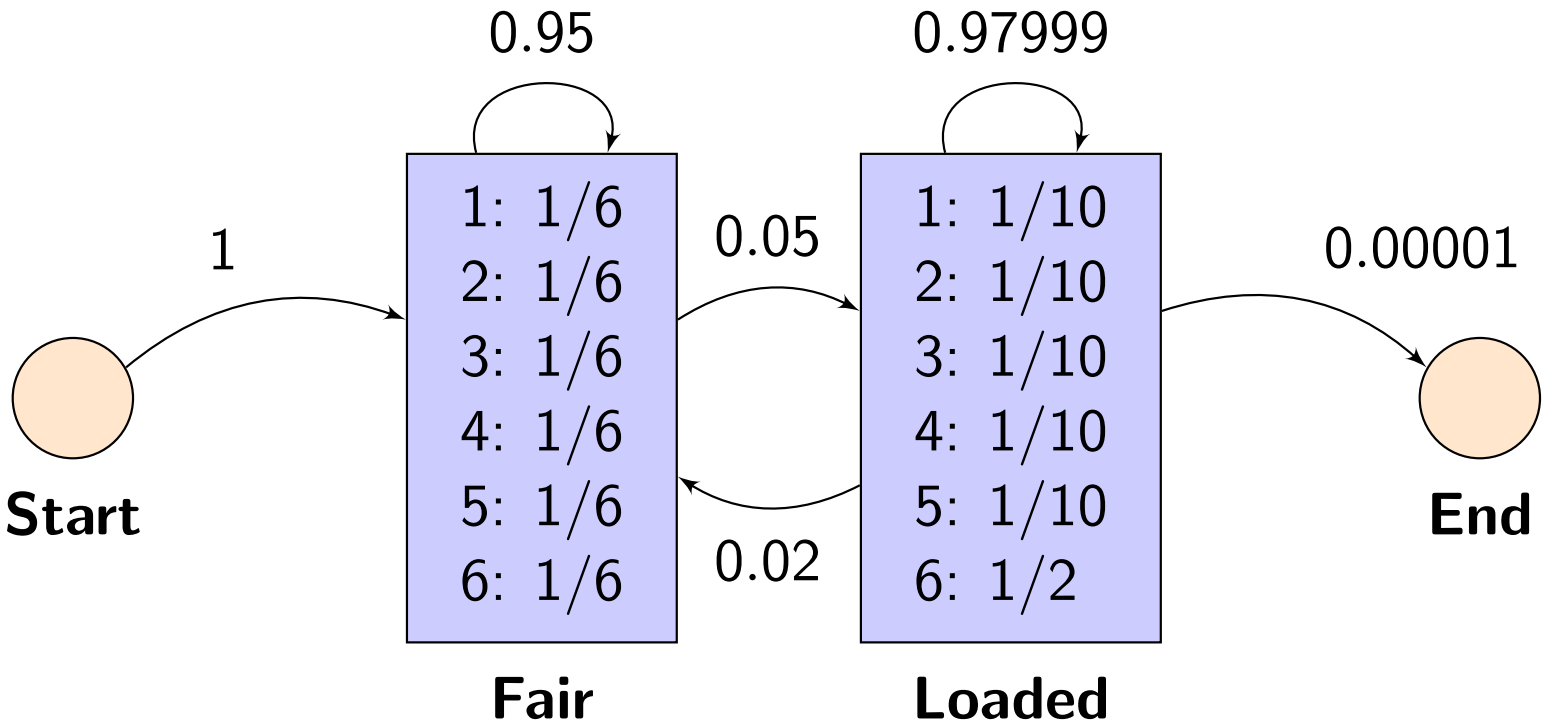
Hidden Markov Models

Trees

A common language?

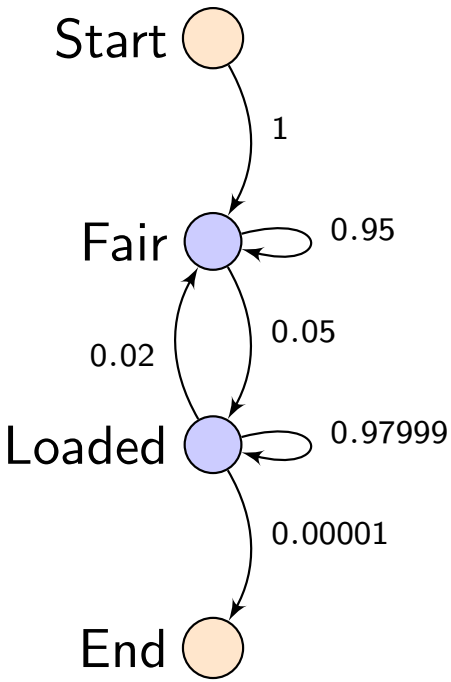


Hidden Markov Models



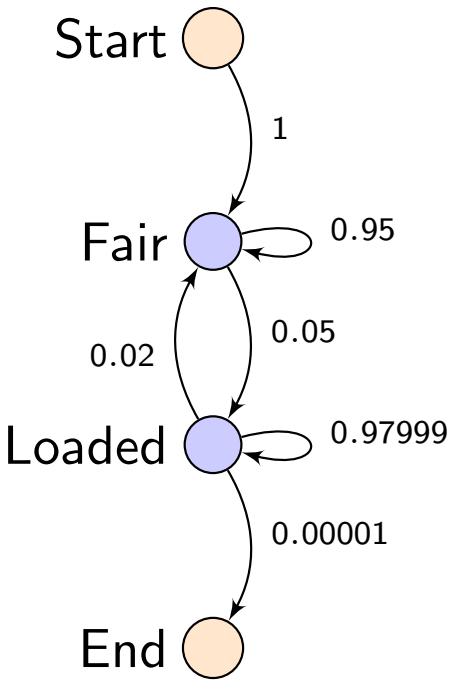
Occasionally dishonest casino

HMMingbird: Hidden Markov Model definition



HMMingbird: Hidden Markov Model definition

HMMingbird




```
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];
}
```

HMMingbird: recursive equations

$$F(s,0) = \begin{cases} 1 & \text{if } s \text{ is start} \\ 0 & \text{otherwise} \end{cases}$$

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

HMMingbird: recursive equations

$$F(s,0) = \begin{cases} 1 & \text{if } s \text{ is start} \\ 0 & \text{otherwise} \end{cases}$$

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

```
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 :
            t.prob * forward(t.start, i - 1))
      * (if s.isend then 1.0 else s.emission{x[i-1]})

result = forward(casino, ``161235456123452'')
```

Trees: Definition language

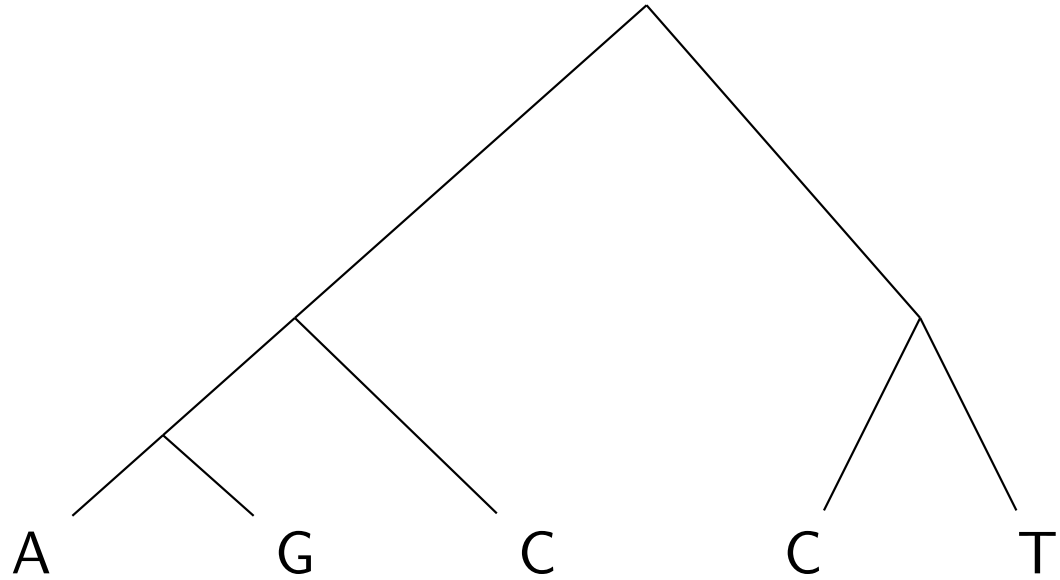
```
define tree phylogenyExample {
   root a;
   node b, c, d;
   leaf e, f, g, h;

   a -> b -> e;
        b -> f;
   a -> c -> d -> g;
        c -> h;
   dataorder e, f, h, g;
}
```

Trees: Simple Recursion

Calculate the tree depth:

```
int d(tree t, treenode[t] n) =
   if n.isleaf
      then 1
   else max(child in n.children : d(child)) + 1

result = d(phylogenyExample)
```

Trees: *Weighted Parsimony*

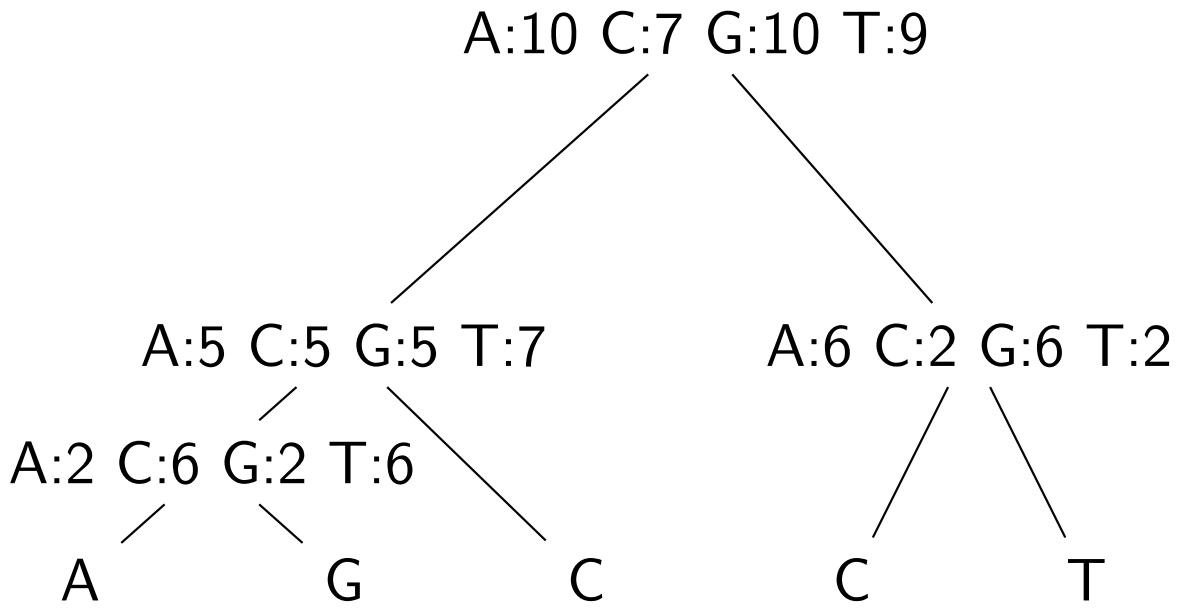


$$w(n,\sigma) = \begin{cases} 0 & \text{if observed character is } \sigma \\ \infty & \text{if observed character different from } \sigma \end{cases}$$

if n is a leaf node and

$$w(n,\sigma) = \min_{\tau \in \Sigma} \{w(\mathrm{R}(n),\tau) + d(\sigma,\tau)\} + \min_{\tau \in \Sigma} \{w(\mathrm{L}(n),\tau) + d(\sigma,\tau)\}$$

Trees: *Weighted Parsimony*



$$w(n,\sigma) = \begin{cases} 0 & \text{if observed character is } \sigma \\ \infty & \text{if observed character different from } \sigma \end{cases}$$

if n is a leaf node and

$$w(n,\sigma) = \min_{\tau\in\Sigma}\{w(\mathrm{R}(n),\tau) + d(\sigma,\tau)\} + \min_{\tau\in\Sigma}\{w(\mathrm{L}(n),\tau) + d(\sigma,\tau)\}$$

Trees: *Weighted Parsimony* Code Example

$$w(n,\sigma) = \begin{cases} 0 & \text{if observed character is } \sigma \\ \infty & \text{if observed character different from } \sigma \end{cases}$$

if n is a leaf node and

$$w(n,\sigma) = \min_{\tau \in \Sigma} \{w(\mathsf{R}(n),\tau) + d(\sigma,\tau)\} + \min_{\tau \in \Sigma} \{w(\mathsf{L}(n),\tau) + d(\sigma,\tau)\}$$

```
score w(tree t, treenode[t] n, treeseq[t] seq, matrix[DNA] d, char[DNA] ch) =
  if n.isleaf then
    if seq[n] == ch then 0
    else infinity
  else
    sum(child in n.children :
          min(nextch in DNA :
            w(child, nextch) + d[ch, nextch]))

datafile = load(treeseq[foo, DNA],"file.txt")
result = map(w, myTree (*) datafile (*) substmatrix)
```

Running example - edit distance

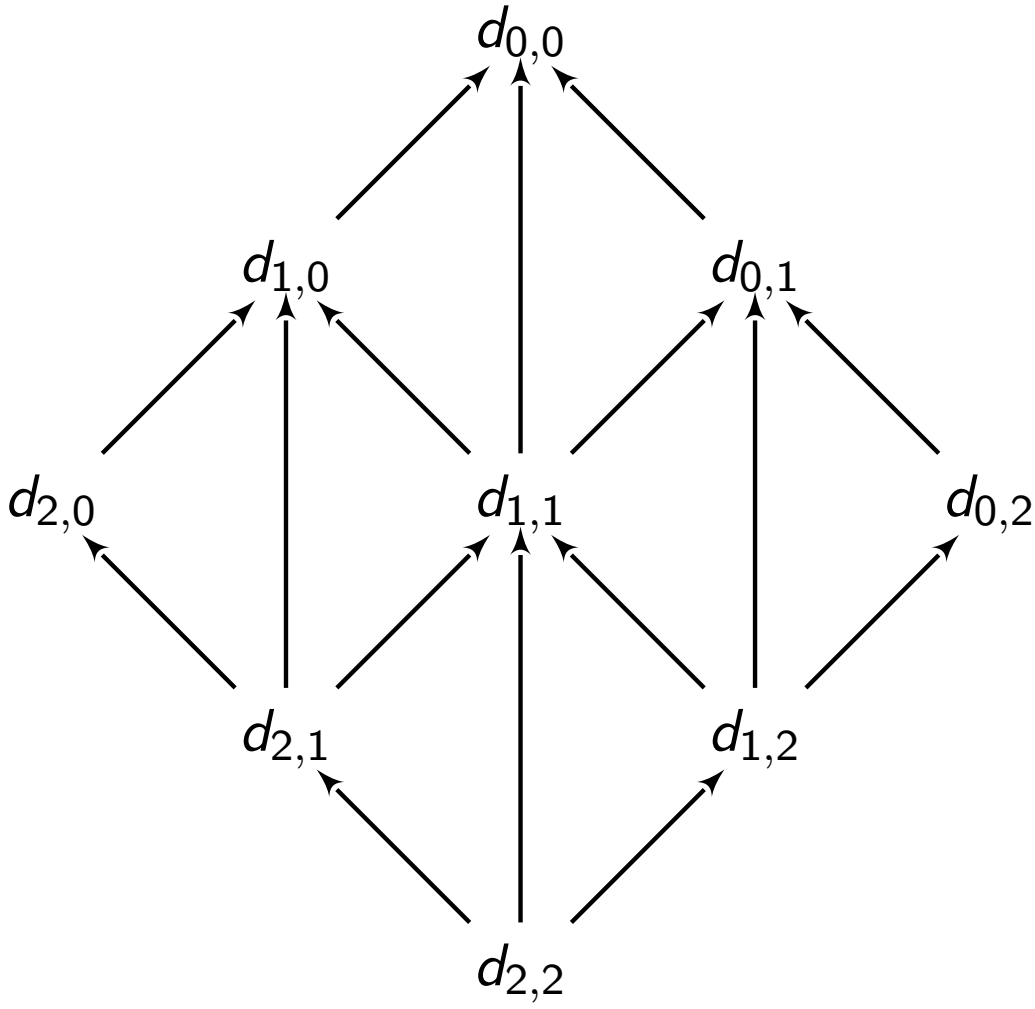
$$d(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ d(i-1,j-1) & \text{if } s[i] = t[j] \\ min \begin{pmatrix} d(i-1,j), \\ d(i,j-1), \\ d(i-1,j-1) \end{pmatrix} + 1 & \text{otherwise} \end{cases}$$

Running example - edit distance

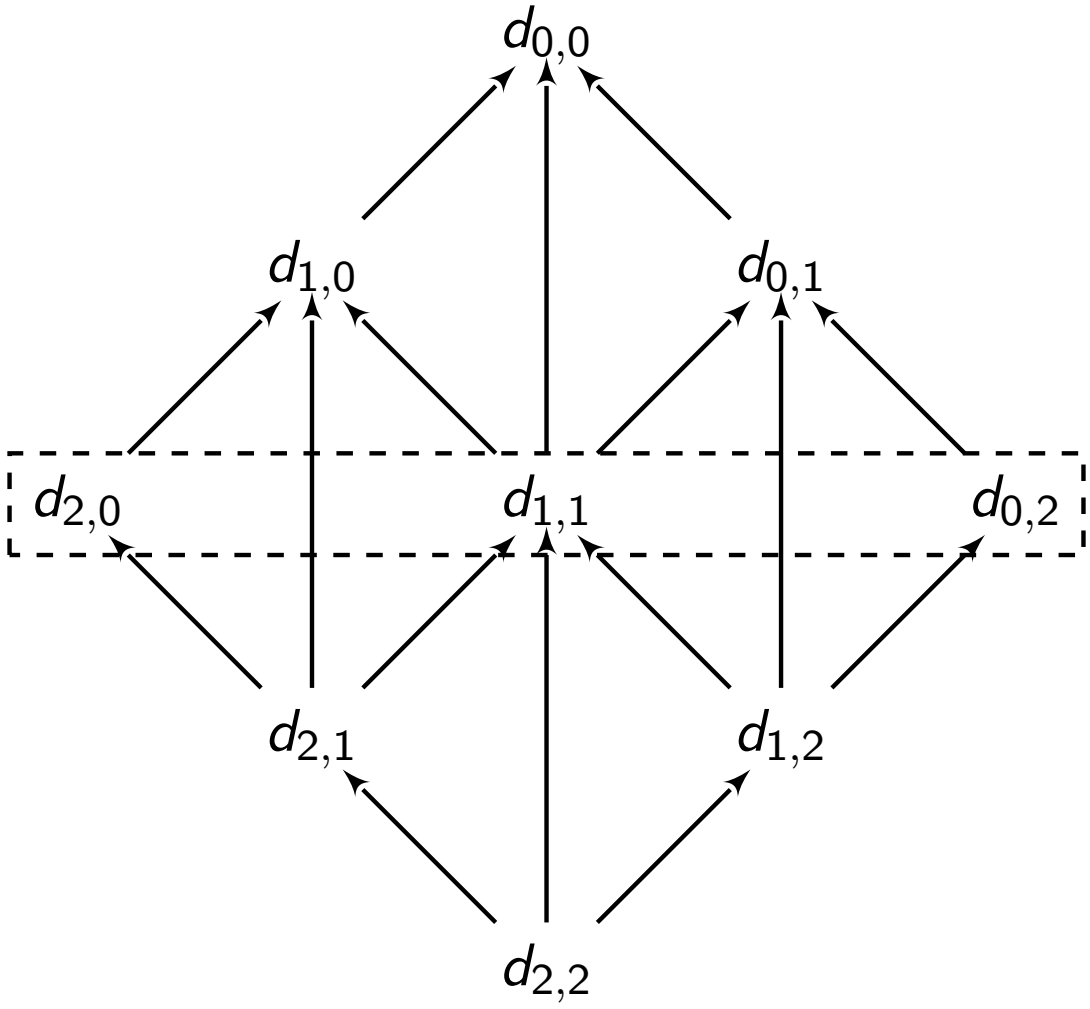
$$d(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ d(i-1,j-1) & \text{if } s[i] = t[j] \\ min \begin{pmatrix} d(i-1,j), \\ d(i,j-1), \\ d(i-1,j-1) \end{pmatrix} + 1 & \text{otherwise} \end{cases}$$

```
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
```

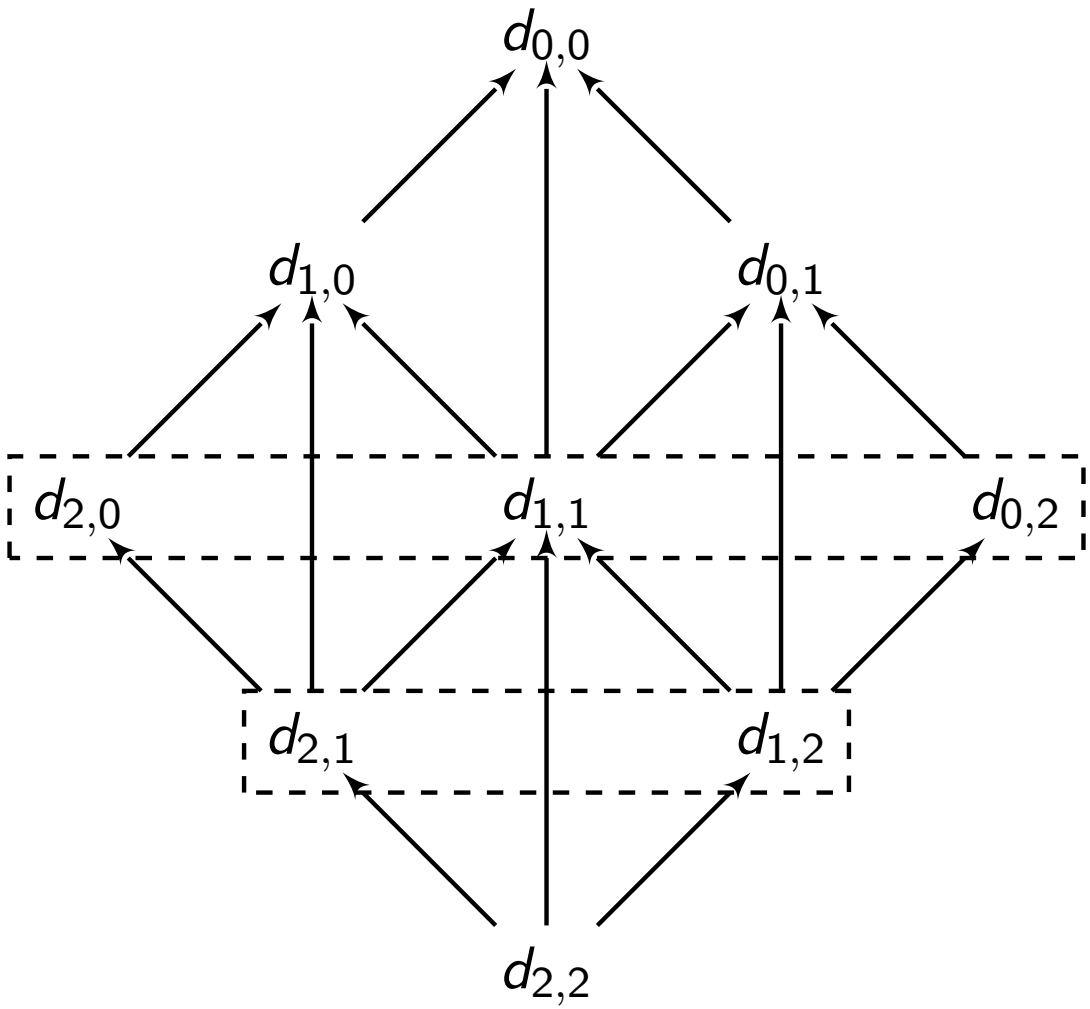
Dependency Graph



Dependency Graph



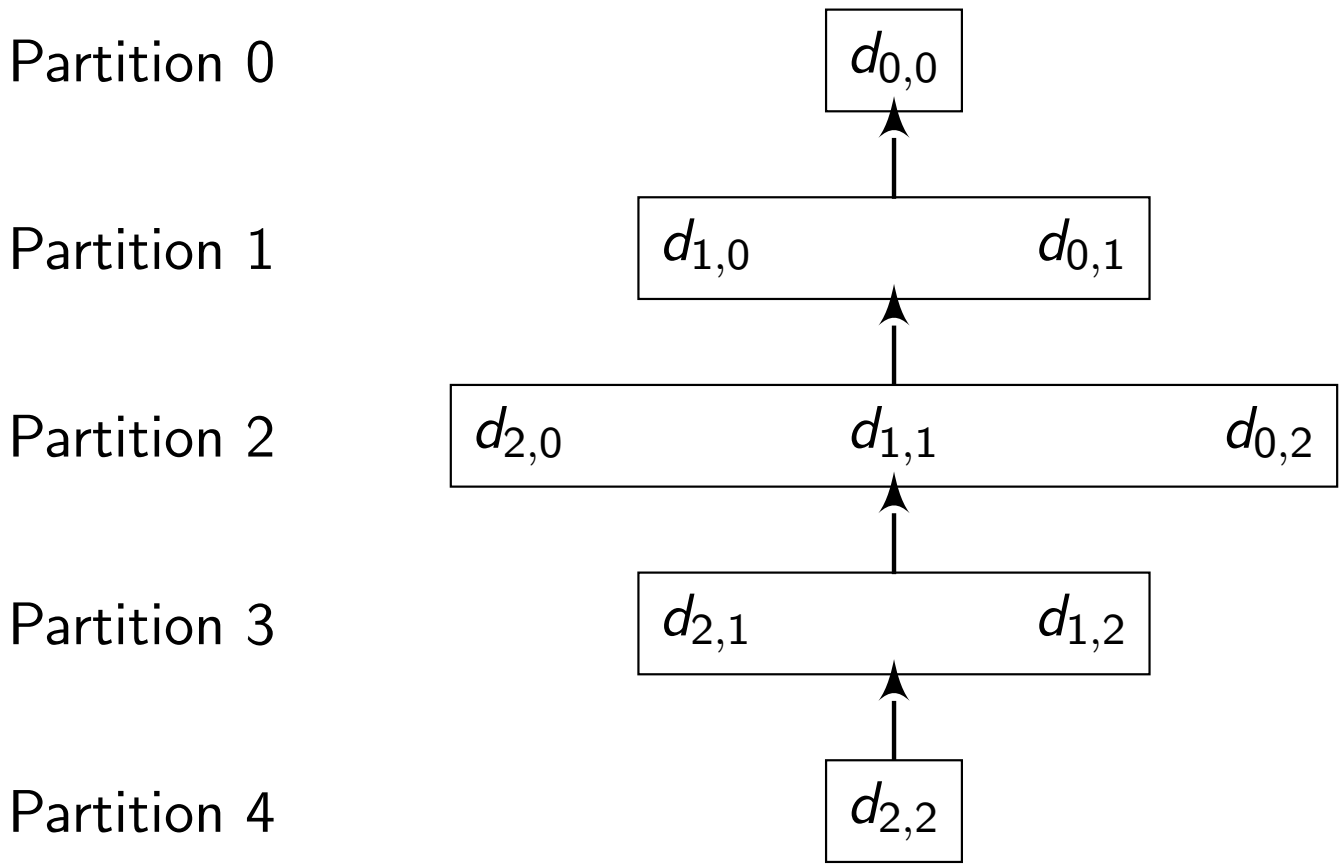
Dependency Graph



Schedule

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

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



The search for a schedule

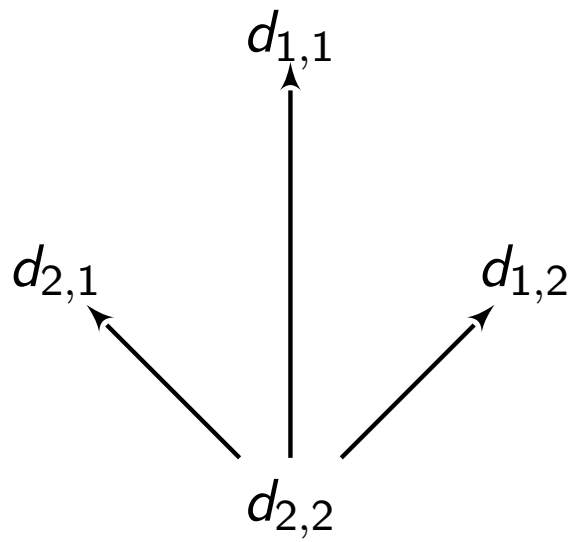
Scheduling Condition

$$d(i_1, j_1) \rightarrow d(i_2, j_2) \implies S_d(i_1, j_1) > S_d(i_2, j_2)$$

The search for a schedule

Scheduling Condition

$$d(i_1, j_1) \rightarrow d(i_2, j_2) \implies S_d(i_1, j_1) > S_d(i_2, j_2)$$



$$S_d(2,2) > S_d(1,0)$$
$$S_d(2,2) > S_d(0,1)$$
$$S_d(2,2) > S_d(1,1)$$

The search for a schedule

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

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

Each recursive call equates to many dependencies

The search for a schedule

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

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

Each recursive call equates to many dependencies

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

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 constraint satisfaction problem (CSP) to choose a *valid* and *efficient* schedule

Identify a schedule 2

Step 2: Construct a constraint satisfaction problem (CSP) to choose a *valid* and *efficient* schedule

Many possible valid schedules - e.g $S_d(i,j) = i + j$, $S_d(i,j) = 2i + 2j$ etc.

Identify a schedule 2

Step 2: Construct a constraint satisfaction problem (CSP) to choose a *valid* and *efficient* schedule

Many possible valid schedules - e.g $S_d(i,j) = i + j$, $S_d(i,j) = 2i + 2j$ etc.

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 constraint satisfaction problem (CSP) to choose a *valid* and *efficient* schedule

Many possible valid schedules - e.g $S_d(i,j) = i + j$, $S_d(i,j) = 2i + 2j$ etc.

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

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

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

Code Generation

We have a schedule + original function - need to generate code.

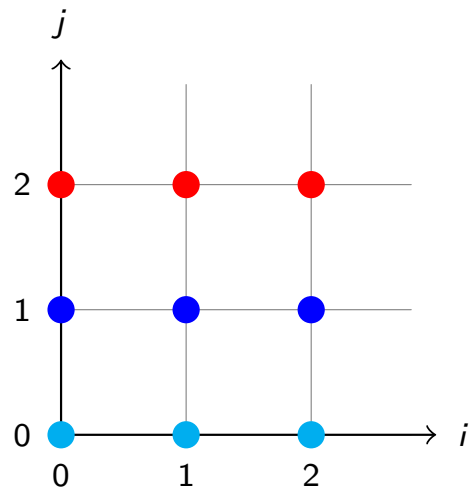
Domain of recursion: $\{0 \leq i \leq n, 0 \leq j \leq m\}$
Schedule: $S_d(i,j) = i + j$

```
// Schedule loop, over partitions
for (p=0;p<=m+n;p++) {
  // Loop over elements in the partitions
  for (??; ??; ??) {
     i = ?;
     j = ?;
     d(i, j);
  }
}
```

Code Generation

We have a schedule + original function - need to generate code.

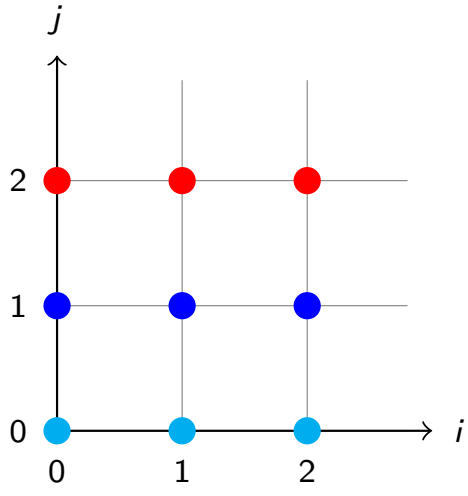
Domain of recursion: $\{0 \leq i \leq n, 0 \leq j \leq m\}$
Schedule: $S_d(i,j) = i + j$

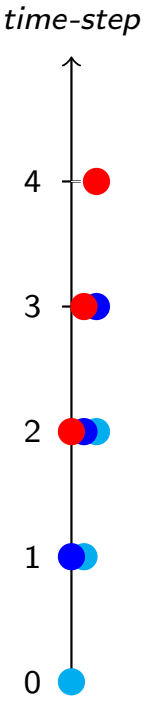


Code Generation

We have a schedule + original function - need to generate code.

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

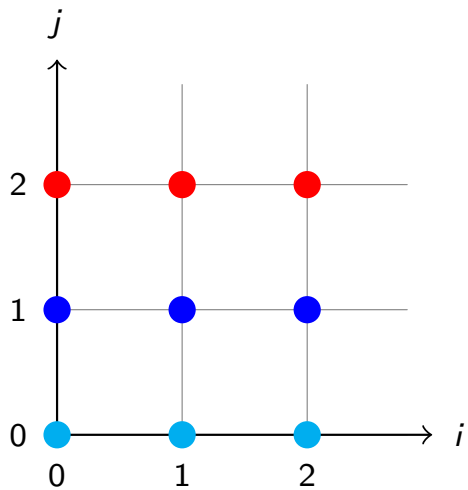


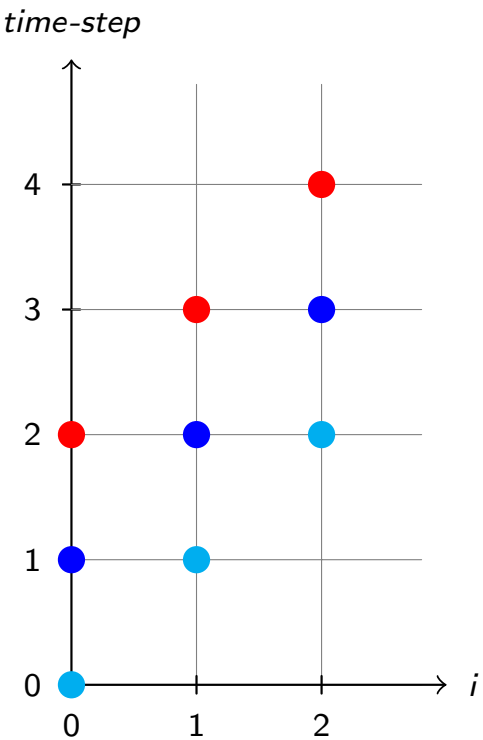


Code Generation

We have a schedule + original function - need to generate code.

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





Code Generation using ClooG

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);
  }
}
```

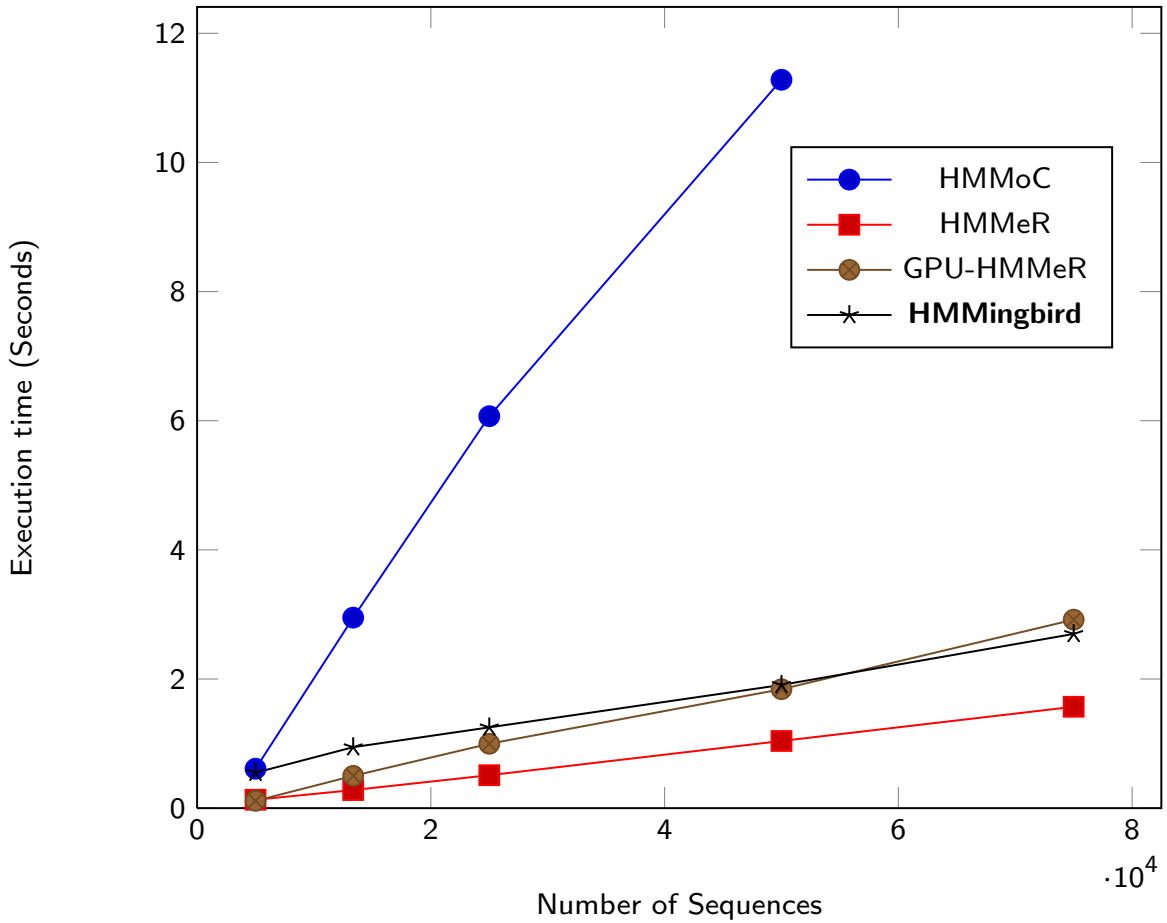
C. Bastoul: Code Generation in the Polyhedral Model is Easier Than You Think. PACT 2004

GPU Code Generation

Block size: *tn* threads

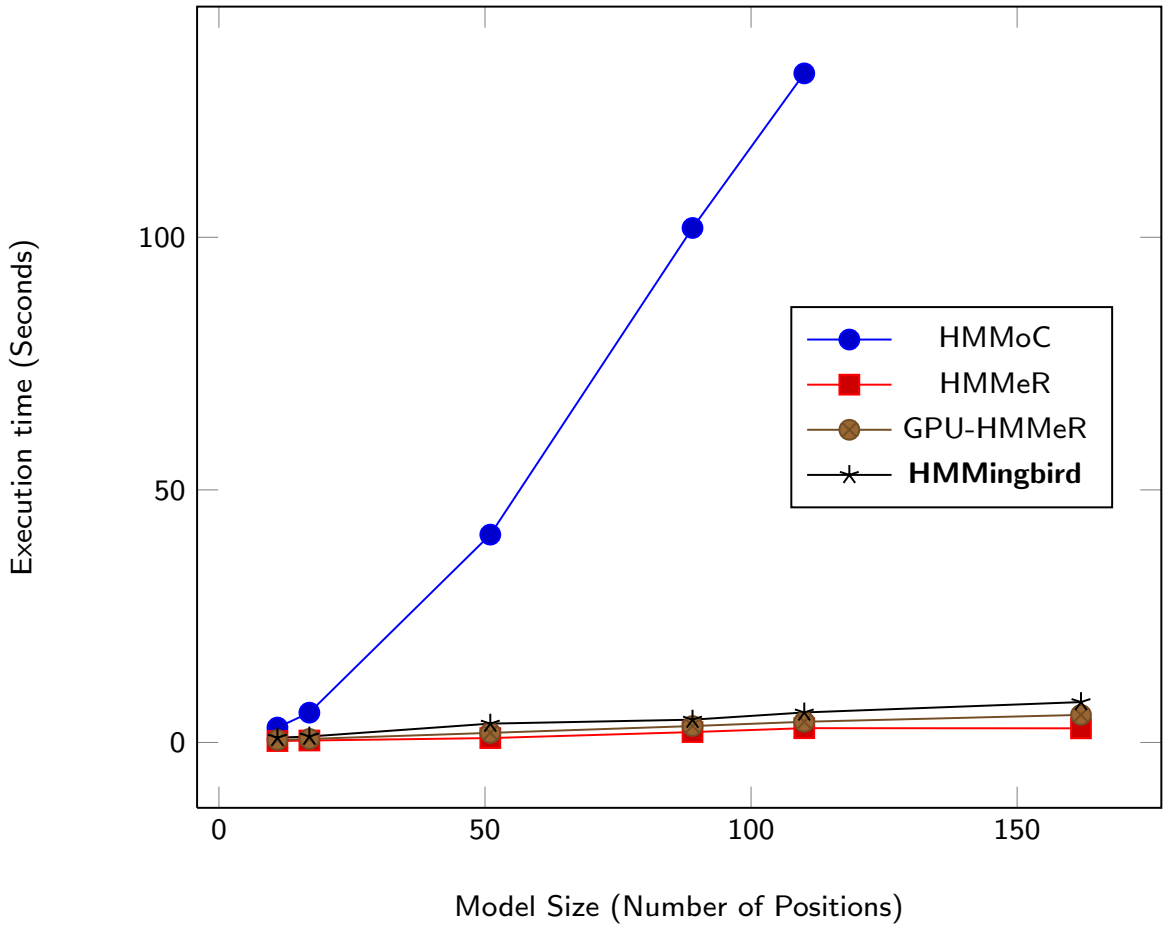
```
parfor threads t in 0..tn {
  for (p=0;p<=m+n;p++) {
    for (i=t+max(0,p-m);i<=min(n,p);i+=tn) {
      j = p - i;
      darr[i,j] = d(i,j);
    }
    sync;
  }
}
```

Performance results



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

Performance results



Performance on a dataset of 13,355 sequences, on models of a varying size.

Conclusion

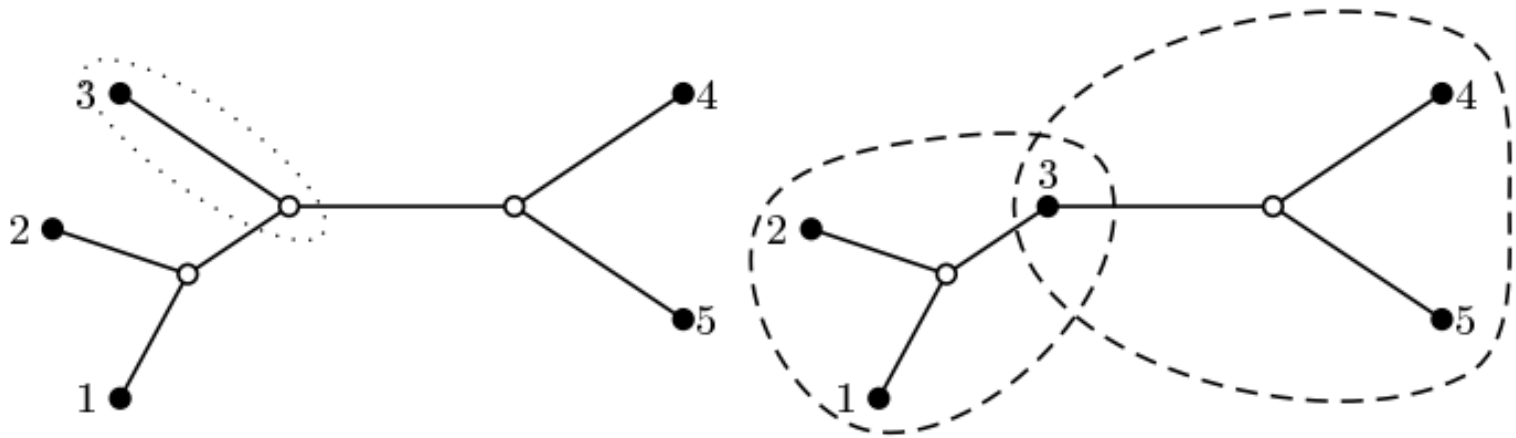
- We can have our cake and eat it too.
- Automatic parallelisation of core recursive language.
- High-level approach can provide both an efficient implementation and happy users (we hope!).

Further Information

Synthesising graphics card programs from DSLs
Luke Cartey, Rune Lyngsoe, Oege de Moor
Programming Language Design & Implementation 2012 (PLDI'12).

luke.cartey@cs.ox.ac.uk

Another example: *Tree Segmentation*



- Phylogenies are essentially steiner trees
- The algorithm aims to segment phylogeny into k-restricted Steiner tree, that is all full components in the tree have at most k leaves
- A full component refers to a subgraph where all leaves are input nodes and all internal nodes are Steiner nodes

Tree Segmentation Recursions

$$
D_u(v) = \begin{cases}
co((u, l(u)), v) + D_{l(u)}(v) + \text{opt} \begin{cases} C_{r(u)}(k-1) + ex((u, r(u)), v) \\ \text{opt}_{w \in R_u} \{D_{r(u)}(w) + ex((u, r(u)), v, w)\} \\ D_{r(u)}(v) + co((u, r(u)), v) \end{cases} & \text{for } v \in L_u \\
co((u, r(u)), v) + D_{r(u)}(v) + \text{opt} \begin{cases} C_{l(u)}(k-1) + ex((u, l(u)), v) \\ \text{opt}_{w \in L_u} \{D_{l(u)}(w) + ex((u, l(u)), v, w)\} \\ D_{l(u)}(v) + co((u, l(u)), v) \end{cases} & \text{for } v \in R_u \\
\sum \begin{cases} \text{opt} \begin{cases} C_{l(u)}(k-1) + ex((u, l(u)), v) \\ \text{opt}_{w \in L_u} \{D_{l(u)}(w) + ex((u, l(u)), v, w)\} \\ D_{l(u)}(v) + co((u, l(u)), v) \end{cases} \\ \text{opt} \begin{cases} C_{r(u)}(k-1) + ex((u, r(u)), v) \\ \text{opt}_{w \in R_u} \{D_{r(u)}(w) + ex((u, r(u)), v, w)\} \\ D_{r(u)}(v) + co((u, r(u)), v) \end{cases} \end{cases} & \text{otherwise}
\end{cases}
$$

$$
C_u(1) = \text{opt}_{v \in L_u \cup R_u} \{D_u(v) + ex((p(u), u), v)\}
$$

$$
C_u(i) = \text{opt} \begin{cases} C_u(i-1) \text{ for } i > 2 \\ \text{opt}_{1 \le j < i} \left\{ C_{l(u)}(j) + C_{r(u)}(i-j) \right\} \end{cases} \quad \text{for } i > 1
$$

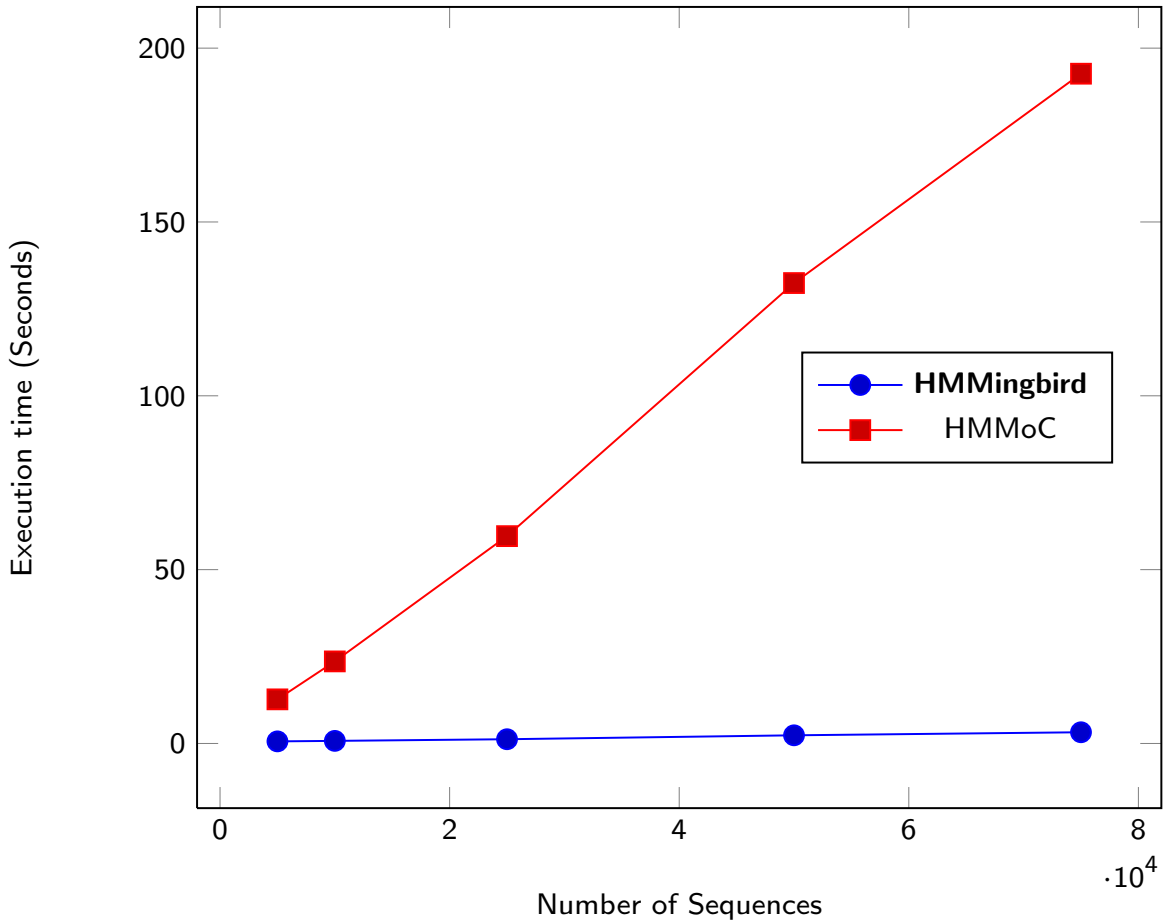
$$
D_u(v) = \begin{cases} 0 & \text{for } u = v \\ \infty & \text{otherwise} \end{cases}
$$

$$
C_u(i) = \begin{cases} 0 & \text{for } i = 1 \\ \infty & \text{for } 2 \le i < k \end{cases}
$$

Tree Segmentation Code

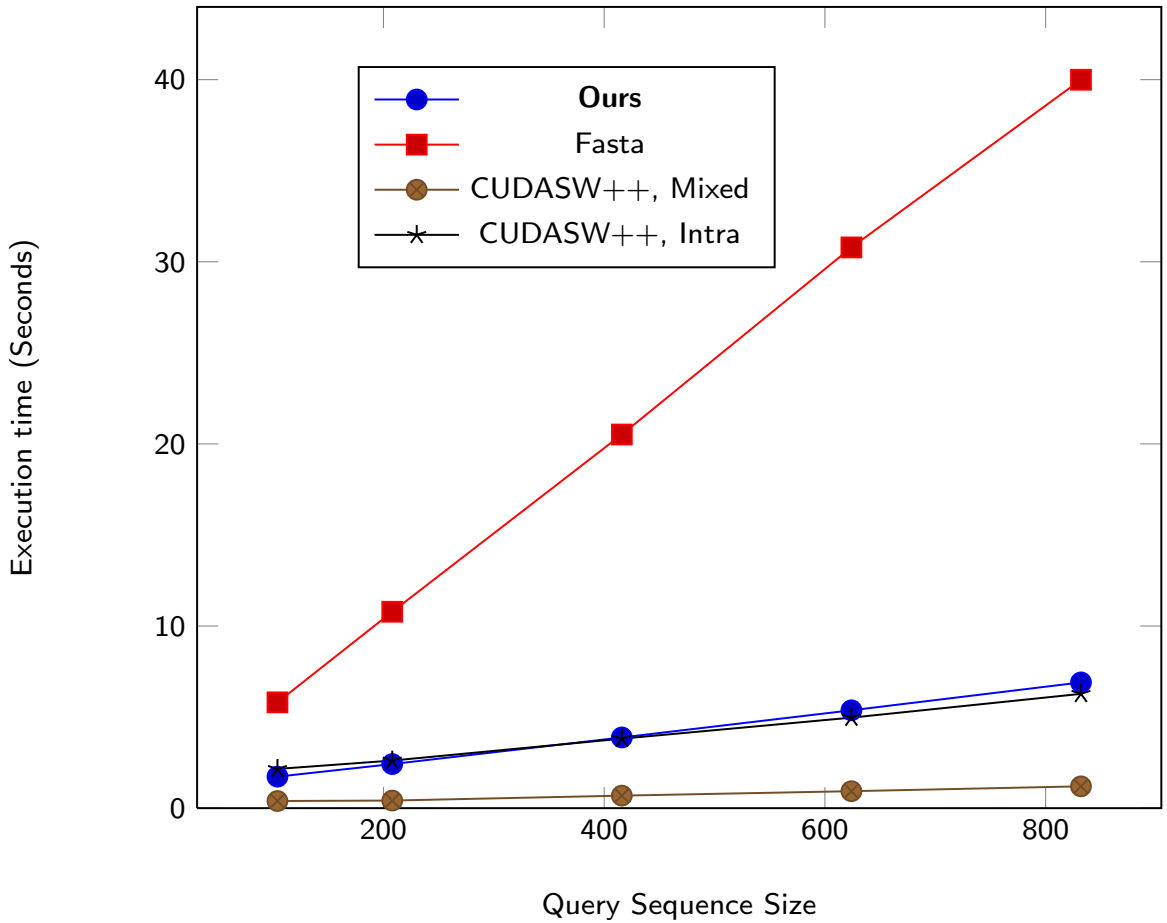
```
int main(tree t) = min(leaf in t.leaves :
edge_contraction(t.root, leaf)) min no_contraction(t.root, k)

int edge_contraction(tree t, node[t] root, node[t] leaf) =
  if root == leaf then
    0
  else if root.left.reccontains[leaf] then
    edge_contraction(root.left, leaf) + below_a_contracted_leaf(root.right, leaf)
  else if root.right.reccontains[leaf] then
edge_contraction(root.right, leaf) + below_a_contracted_leaf(root.left, leaf)
  else
    below_a_contracted_leaf(root.left, leaf) + below_a_contracted_leaf(root.right, leaf)


int below_a_contracted_leaf(tree t, node[t] child, node[t] leaf) =
   (no_contraction(child, k-1) + 1)
 min
   (min(anotherleaf in child.recleaves : edge_contraction(child, anotherleaf) ))
 min
   (edge_contraction(child, leaf) + 1)


int no_contraction(tree t, node[t] root, int leaves_remaining) =
  if leaves_remaining == 1 then
     min(leaf in root.recleaves : edge_contraction(root, leaf))
  else
     min(leaf in root.recleaves : edge_contraction(root, leaf)) min min(j in 1 ... i-1 :
     no_contraction(root.left, j) + no_contraction(root.right, i-j))
```

Performance results - Gene finder



Gene-finding performance on varying sequence sizes.

Performance results - Smith Waterman



Smith-Waterman performance on varying query sequence sizes for a database of 75,000 sequences.