Automatically Deriving Cost Models for Structured Parallel Programs using Types and Hylomorphisms

David Castro, Kevin Hammond and Susmit Sarkar

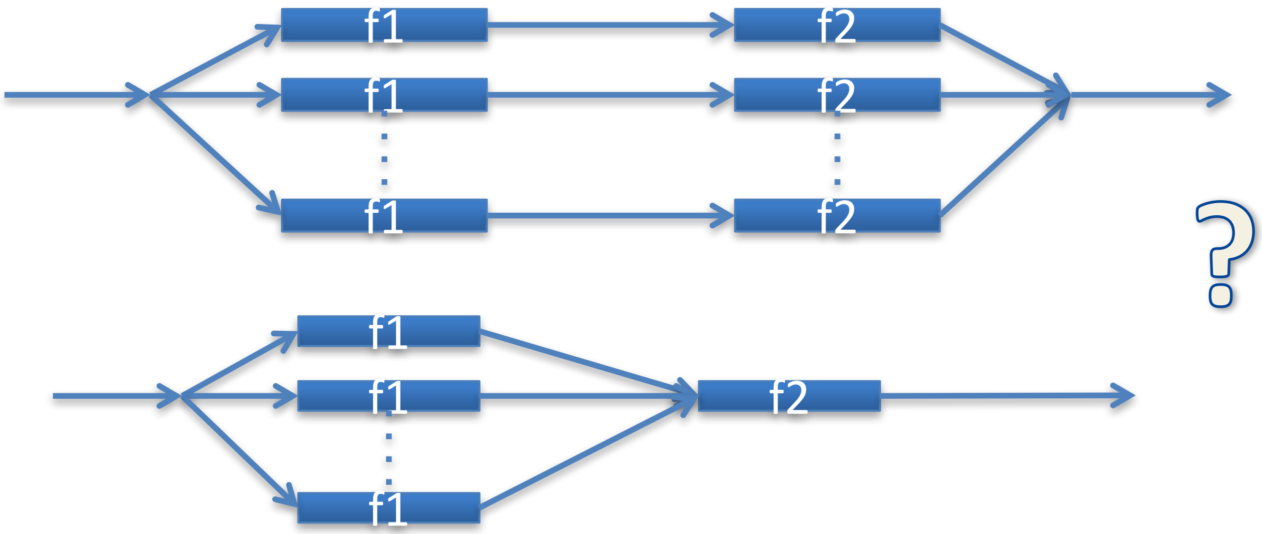
University of St Andrews, UK

T: *@rephrase_eu, @khstandrews*
E: *kevin@kevinhammond.net*
W: http://www.rephrase-ict.eu
http://www.paraformance.com

Motivation

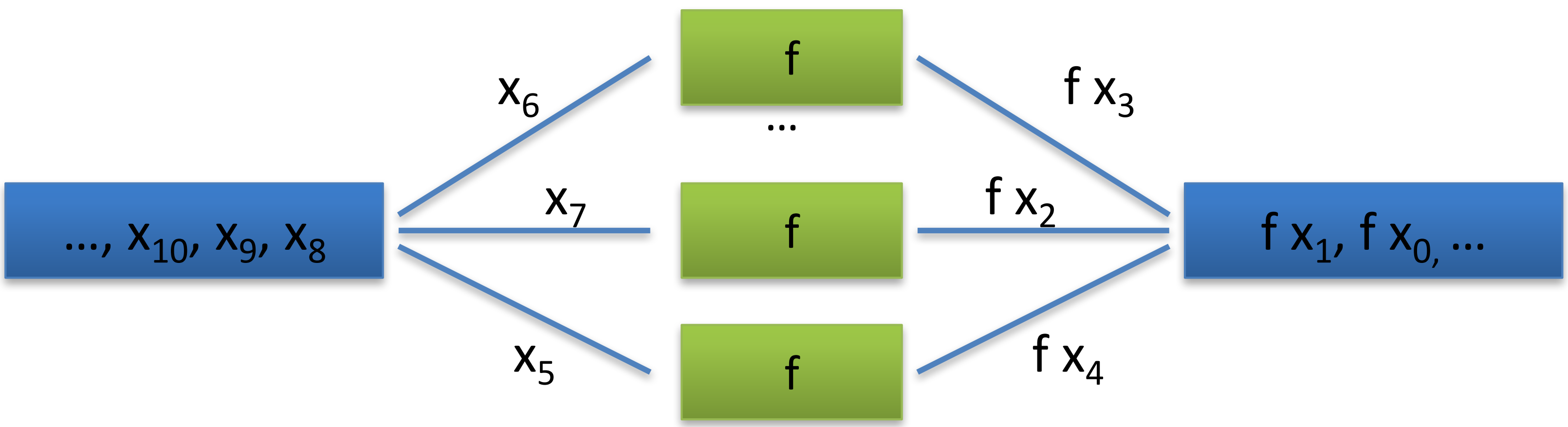
- Parallel patterns are great
  - **BUT we need to choose the best implementation**
  - *For a specific (heterogeneous) parallel architecture*

- We need a way to reason about parallel structure
  - ✓ Correctness of transformations (done! **ICFP2016**)
  - ❑ Reasoning about performance (in progress)



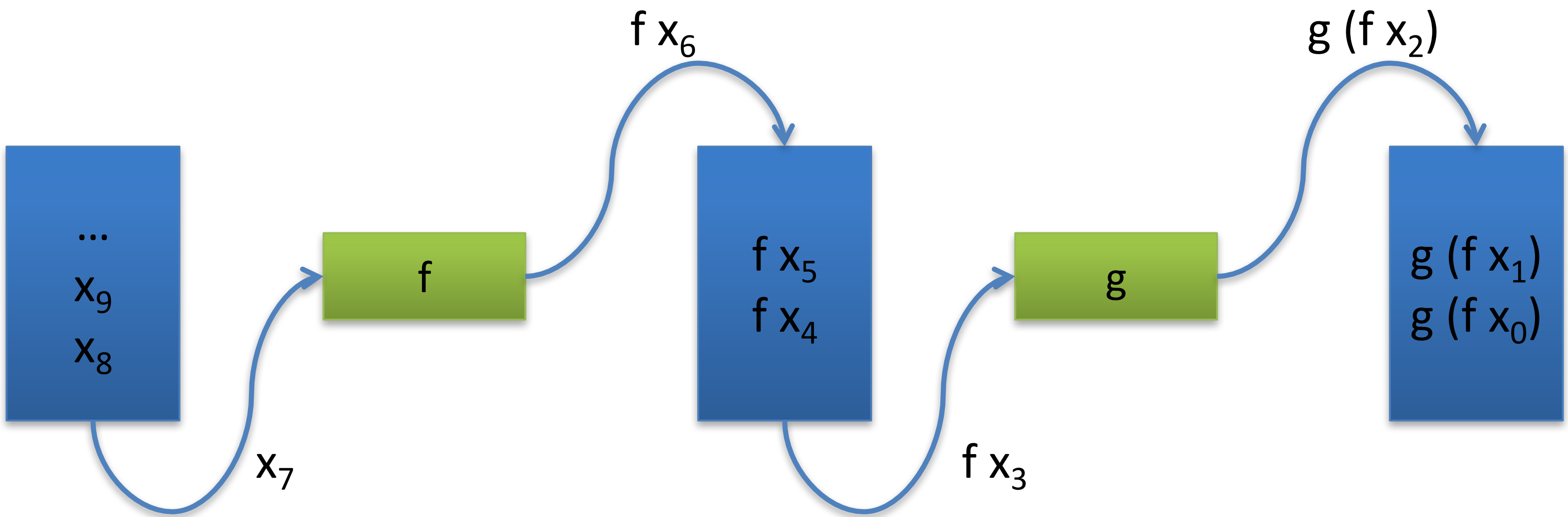
Example Skeleton: Parallel Task Farms

- **Task Farms use a fixed number of workers (farm *n* *f*)**
  - Each worker applies the same operation (*f*)
  - *f* is applied to each of the inputs in a stream.



Example Skeleton: Parallel Pipeline

- **Parallel pipelines compose two operations (*f* || g)**
  - over the elements of an input stream
  - *f* and **g** are run in parallel



Example: Parallel Image Merge

University of St Andrews

Image merge (im) composes mark and merge

im : List (Img, Img) -> List Img
im = map (merge ∘ mark)

There are many alternative parallel implementations
- **even just** using farms (farm) and pipelines (||)

im_1 = farm n (fun (merge ∘ mark))
im_2 = farm n (fun mark) || farm m (fun merge)
im_3 = farm n (fun mark) || fun merge
…

So, why types?

- According to the types community:
  - **Soundness**: "Well-typed programs cannot go wrong"
  - **Documentation**: "Type signatures provide valuable docs."

- The benefits we are really interested in:
  - **Soundness**:
    - "Well-typed programs can be parallelised as described by the types"
  - **Documentation**:
    - "Type-level parallel structures clearly separate structure & functionality"
  - **Reusing well-understood techniques**.
    - E.g. algorithms for type unification and inference.

Selecting an Implementation

Decorate the function type with IM(n,m)

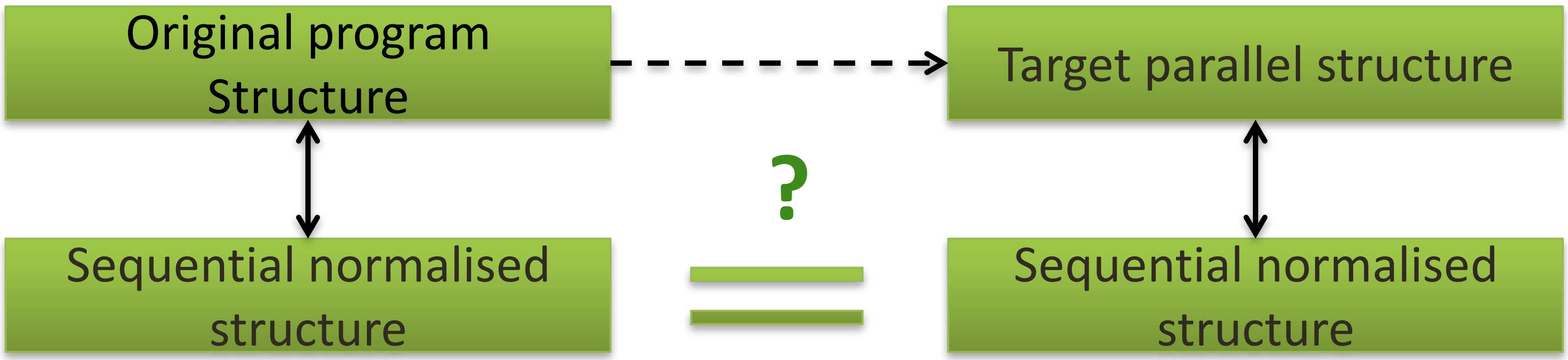
```
im : IM(n,m) ~ List (Img, Img) -> List Img
im = map (merge ∘ mark)

IM(n,m) = FARM n (FUN A) || FARM m (FUN A)
```

The type system now automatically selects

im_2 = farm n (fun mark) || farm m (fun merge)

We can *guarantee* that this is functionally equivalent to *im*

Introducing/Transforming Parallel Patterns



How do we decide semantic equivalences?

University of St Andrews

- We can use the laws and properties of ***hylomorphisms***!

- Hylomorphisms are a generalisation of a divide and conquer.

$$
\begin{aligned}
&\mathrm{hylo}_F\ g\ h = f \\
&\quad \text{where } f = g \circ F\ f \circ h
\end{aligned}
$$

  - "h" splits the input into a structure "F", then recursive calls are mapped in structure "F", the results are combined by "g".

- Algorithmic skeletons can be described as instances of hylomorphisms

Hylomorphism Example

```
type T A = Empty | (A, List A, List A)

quicksort : List A -> List A
quicksort = hylo_T merge split

merge : T A  -> List A
merge = …

split : List A -> T A
split = …
```

Introducing Parallelism

University of St Andrews

We start with a streaming sequential version

quicksort : List (List A) -> List (List A)
quicksort = map_{List} ($hylo_T$ merge split)

To create a task farm and pipeline version, just change the type!

quicksort : PAR_L (FARM n _ || _)~
List (List A) -> List (List A)

To create a parallel divide-and-conquer version, change the type again!

quicksort : PAR_L ($DC_{n,T}$ A A) ~
List (List A) -> List (List A)

RePhrase

ParaFormance™

Base Semantics

- Defined using well-known recursion schemes:
  - map (replication, map_F)
  - fold ("catamorphism", $cata_F$)
  - unfold ("anamorphism", ana_F)
- plus sequential composition, $\circ$

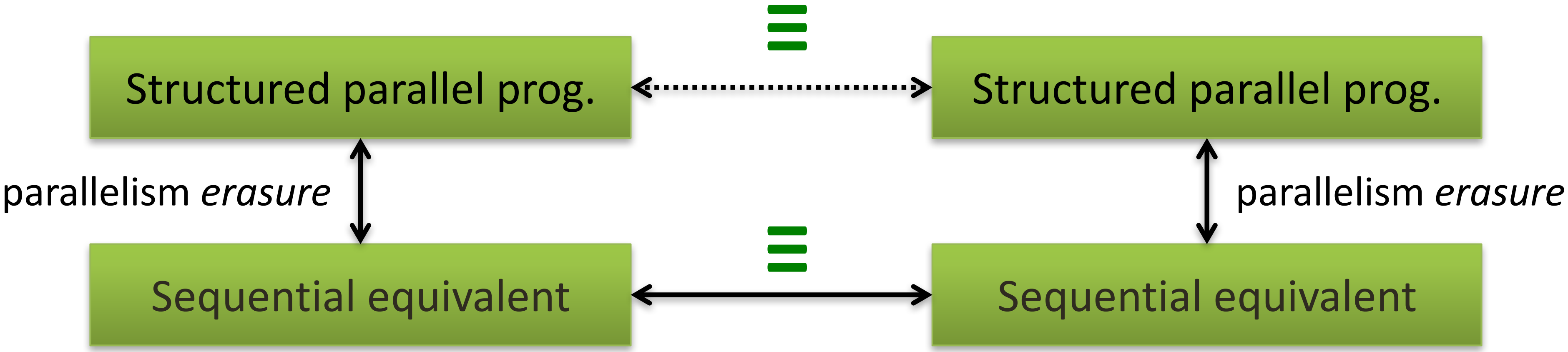
$$
\begin{array}{ll}
S[\![p : T\ A \text{ -> } T\ B]\!] : [\![A \text{ -> } B]\!] & \\
S[\![fun_T\ f]\!] & = env(f) \\
S[\![p_1 \parallel p_2]\!] & = S[\![p_2]\!] \circ S[\![p_1]\!] \\
S[\![farm\ n\ p]\!] & = S[\![p]\!] \\
S[\![dc_{n,T,F}\ f\ g]\!] & = cata_F\ (env(f)) \circ ana_F\ (env(g))
\end{array}
$$

$$
\begin{array}{lcl}
[\![p\ :\ T\ A \rightarrow T\ B]\!] & : & [\![T\ A \rightarrow T\ B]\!] \\
[\![p]\!] & = & map_T\ \mathcal{S}[\![p]\!]
\end{array}
$$

Deciding Semantic Equivalences

University of St Andrews

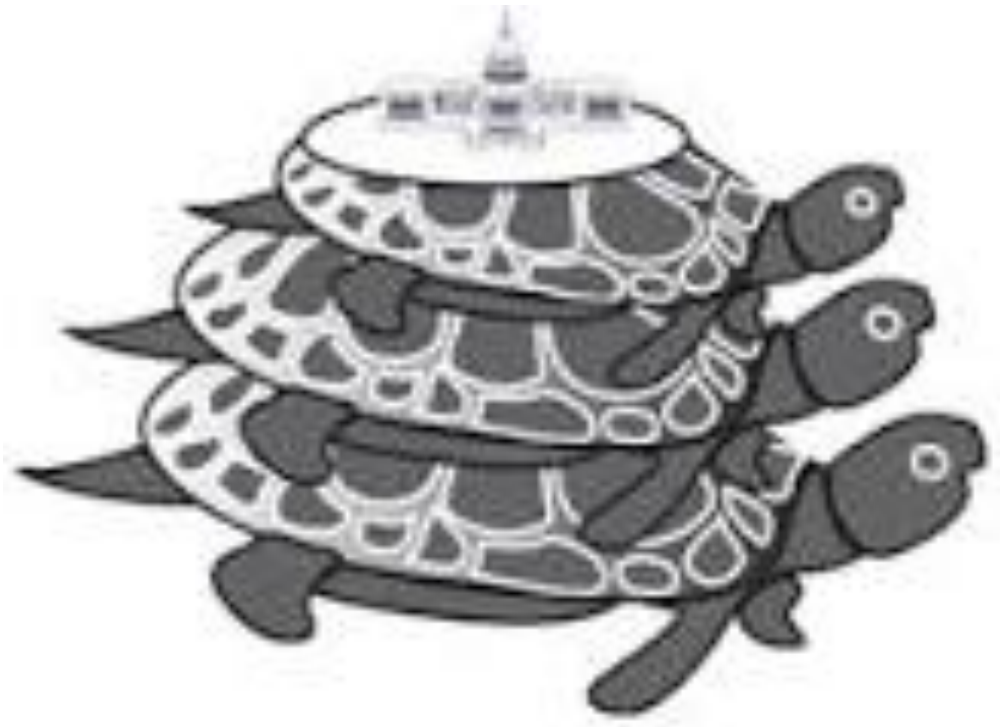
- Equivalence of parallel programs is reduced to equivalence of recursion schemes.



Recursion Schemes are Hylomorphisms!

University of St Andrews

$$
\begin{array}{lcl}
T\ A & = & \mu(F\ A) \\
map_T\ f & = & hylo_{F\ A}\ (in_{F\ B} \circ (F\ f\ id))\ out_{F\ A}, \\
 & & \quad \text{where } A = dom(f) \text{ and } B = codom(f) \\
cata_F\ f & = & hylo_F\ f\ out_F \\
ana_F\ f & = & hylo_F\ in_F\ f
\end{array}
$$



Inferring Parallel Structures

University of St Andrews

We can leave holes in the types

IM(n,m) = _ || FARM m _

Type unification replaces _ with any suitable parallel structure

IM(n,m) = min *cost* (_ || FARM m _)

Type unification replaces _ with the parallel structure that *minimises* the cost model!

Example: Cost Model for Task Farms

University of St Andrews

$$\mathtt{qfarm}(n, \mathcal{P})(Q_0, Q_1) = \overbrace{\mathcal{P}(Q_0, Q_1) \parallel \ldots \parallel \mathcal{P}(Q_0, Q_1)}^{n \text{ times}}$$

$$\Downarrow$$

This cost depends on the number of contending threads

If $\mathcal{P}$ takes time $\mathcal{T}$, then the cost of each $\mathcal{P}(Q_0, Q_1)$ is

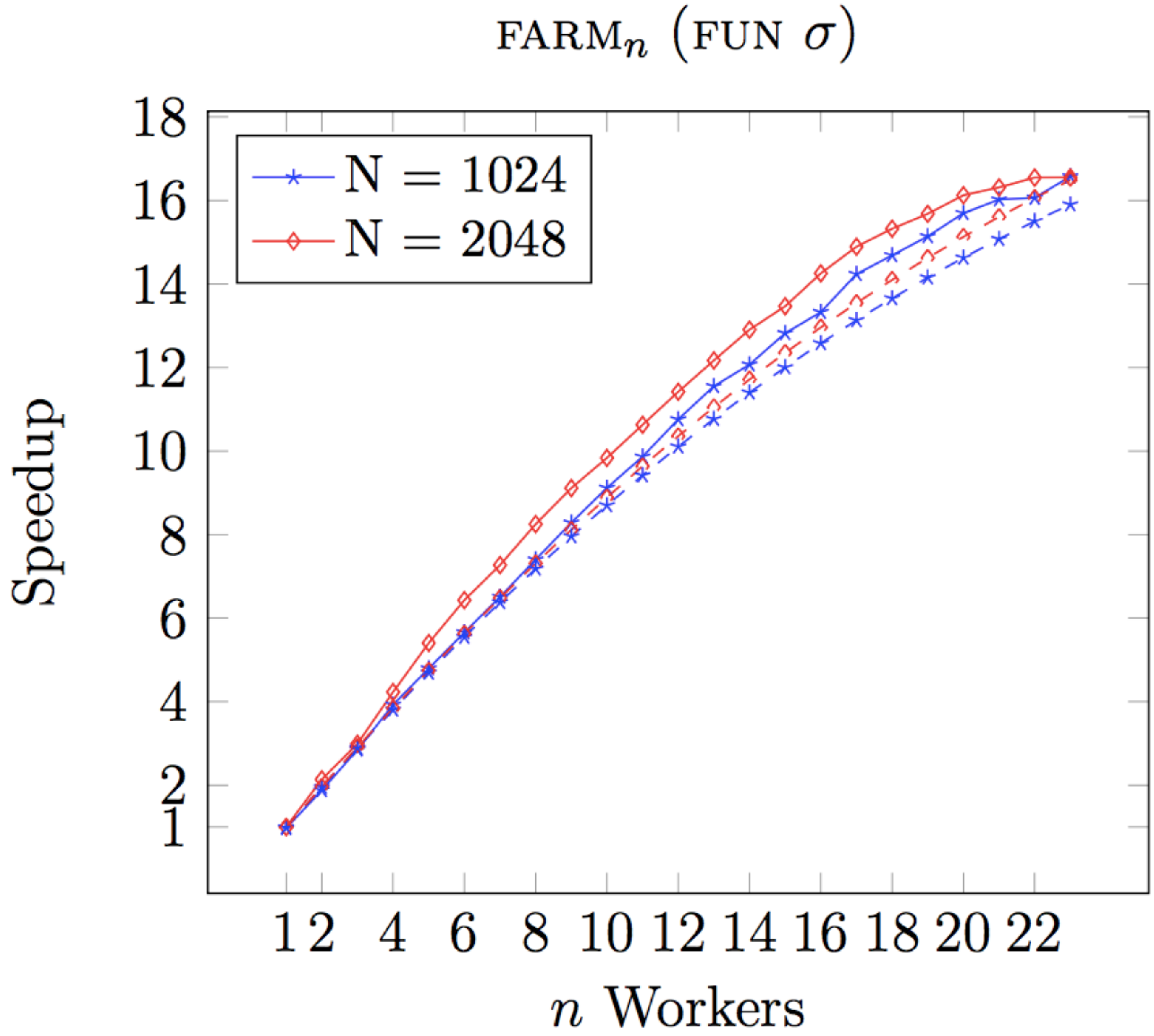
$$\mathcal{T} + \mathcal{T}_{dequeue}(Q_0) + \mathcal{T}_{enqueue}(Q_1).$$

If $\mathcal{P}$ produces p number of outputs, then the task farm produces $n \times p$ number of outputs, so the resulting cost needs to be divided by $n \times p/p$, or n:

$$\frac{\mathcal{T} + \mathcal{T}_{dequeue}(Q_0) + \mathcal{T}_{enqueue}(Q_1)}{n}.$$

RePhrase ParaFormance™

Predicting Parallel Execution Costs



Matrix multiplication,
NxN matrices
24-core AMD Opteron

Predicting Parallel Execution Costs

$$(\text{FARM}_{n_1}(\text{FUN } \sigma_1) \parallel (\text{FARM}_{n_2}(\text{FUN } \sigma_2))$$

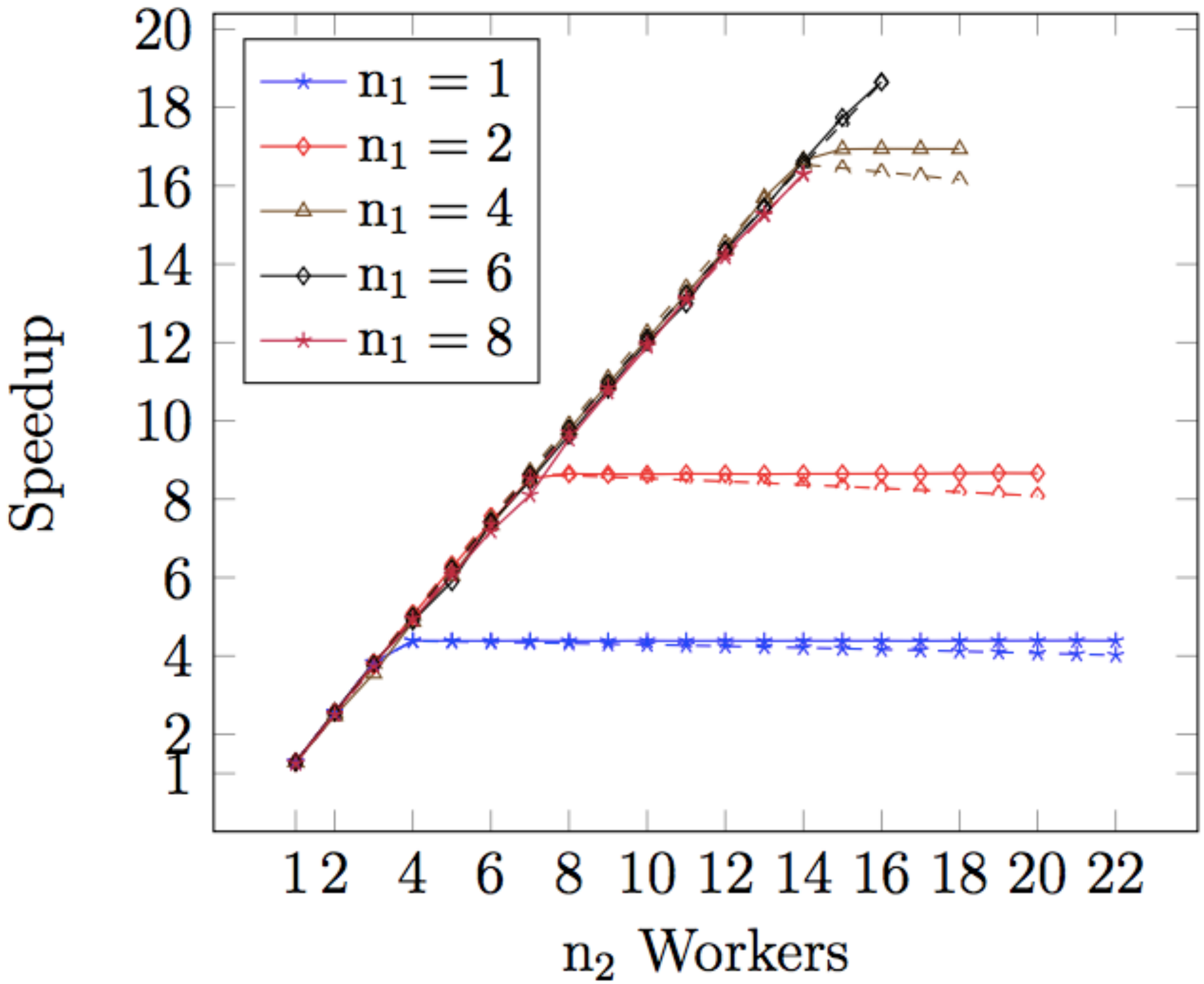


Image Convolution
24-core AMD Opteron

Alternative Parallel Structures

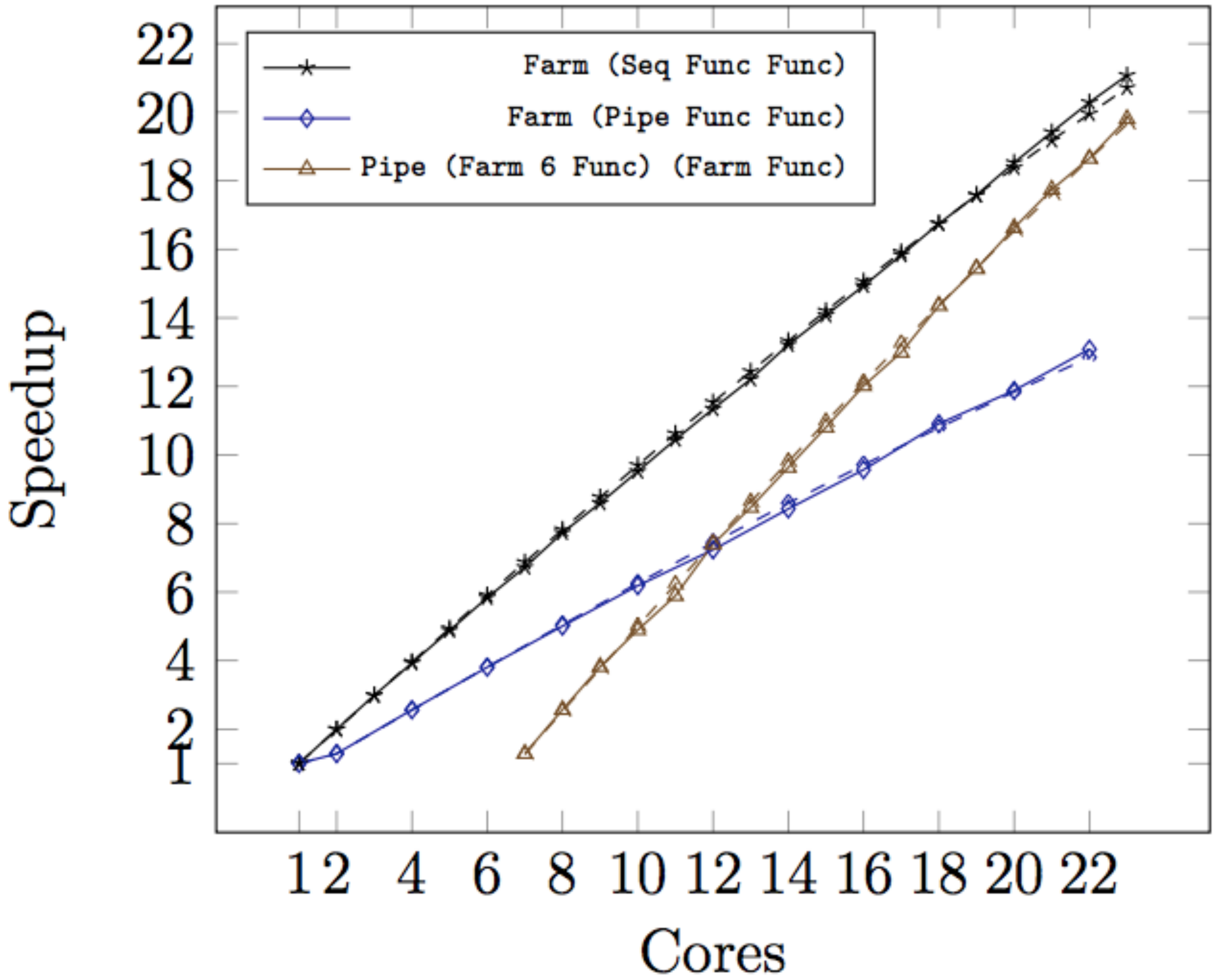


Image Convolution
24-core AMD Opteron

Alternative Parallel Structures

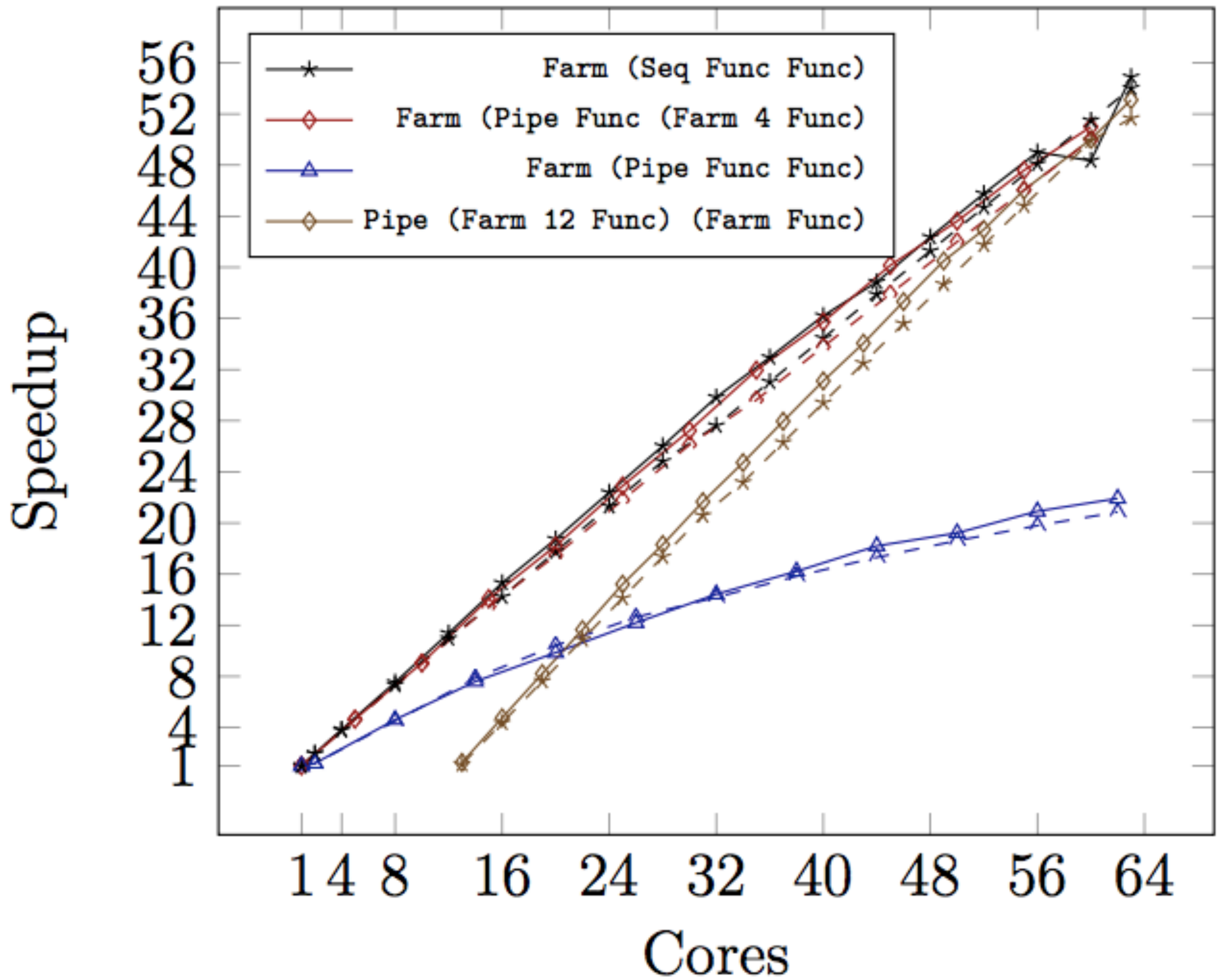


Image Convolution
64-core Intel Xeon

University of St Andrews

ParaFormance™

Conclusions

University of St Andrews

- Deriving costs of parallel structures from an operational semantics is very powerful:
  - Automatically derive a **cost equation** from an "implementation"
  - Compile-time information about run-time behaviour based on a simple and easy to understand model.
  - When combined with our previous work (ICFP 2016), we can automatically rewrite programs to **minimize costs**

- Our cost model accurately predicts lower bounds on speedups

- We can choose between alternative parallel implementations
  - different patterns
  - CPU/GPU, manycore/multicore

RePhrase

ParaFormance™

Future Work

- Other patterns, e.g. stencil and bulk synchronous parallelism
- More general recursion patterns:
  - e.g. adjoint folds or conjugate hylomorphisms (Hinze)
- Apply to real languages (e.g. Haskell, Erlang)
  - Build a full implementation

RePhrase Project:Refactoring Parallel Heterogeneous Software – a Software Engineering Approach (ICT-644235), 2015-2018, €3.5M budget

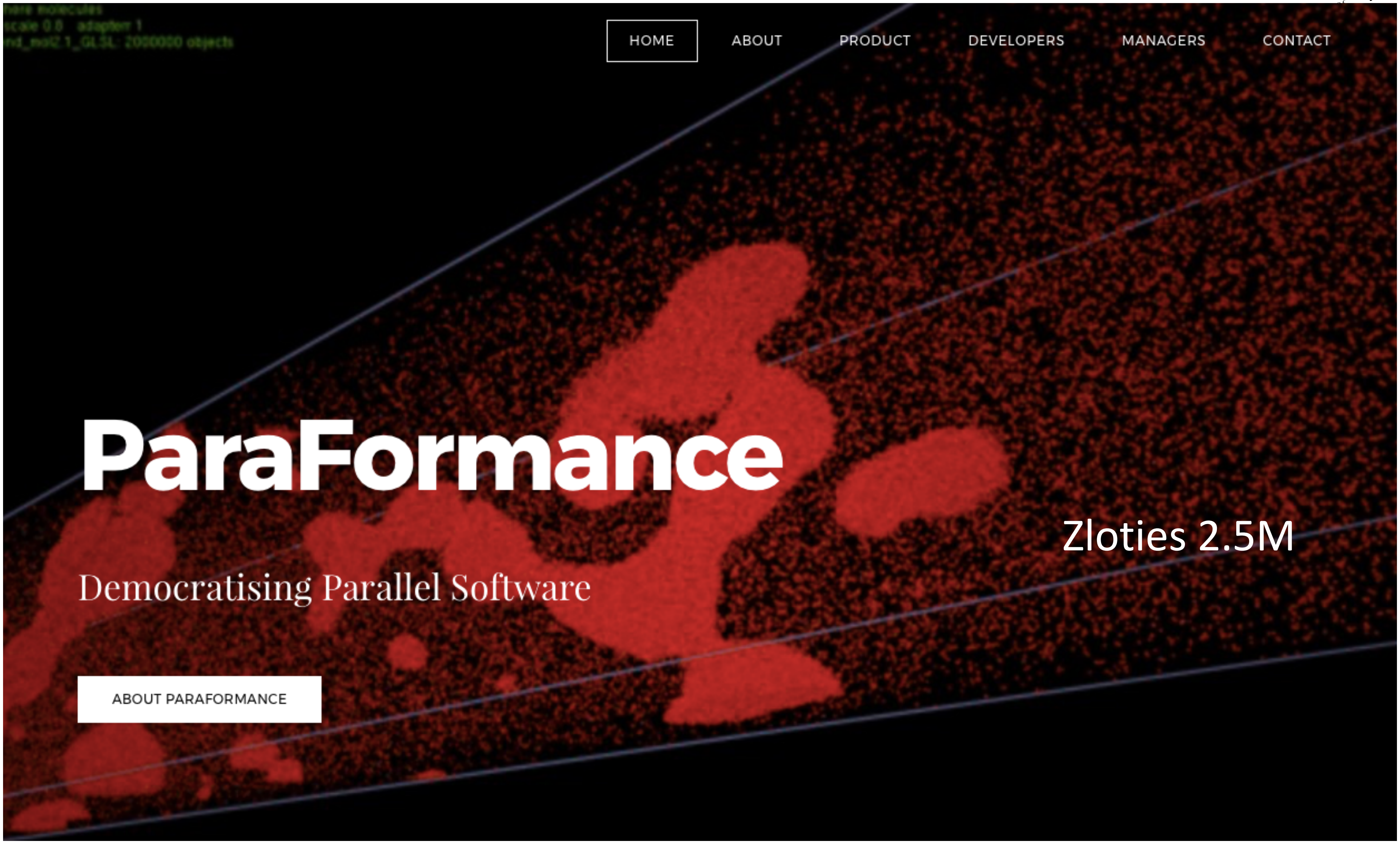
8 Partners, 6 European countries

UK, Spain, Italy, Austria, Hungary, Israel

Coordinated by @khstandrews

Bringing Functional Ideas to the Masses!

University



RePhrase

ParaFormance™

University of St Andrews

THANK YOU!

http://rephrase-ict.eu

http://paraphrase-ict.eu

@rephrase_eu

Type System

$$\frac{\rho(f) = A \to B}{\vdash f \;:\; A \xrightarrow{\mathrm{A}} B}
\qquad
\frac{\begin{array}{c}\vdash e_1 \;:\; B \xrightarrow{\sigma_1} C \\ \vdash e_2 \;:\; A \xrightarrow{\sigma_2} B\end{array}}{\vdash e_1 \circ e_2 \;:\; A \xrightarrow{\sigma_1 \circ\, \sigma_2} C}
\qquad
\frac{\begin{array}{c}\vdash e_1 \;:\; F\,B \xrightarrow{\sigma_1} B \\ \vdash e_2 \;:\; A \xrightarrow{\sigma_2} F\,A \qquad G = \mathsf{base}\ F\end{array}}{\vdash \mathsf{hylo}_F\ e_1\ e_2 \;:\; A \xrightarrow{\mathrm{HYLO}_G\ \sigma_1\ \sigma_2} B}
\qquad
\frac{\begin{array}{c}\vdash p \;:\; T\,A \xrightarrow{\sigma} T\,B \\ F = \mathsf{base}\ T\end{array}}{\vdash \mathsf{par}_T\ p \;:\; T\,A \xrightarrow{\mathrm{PAR}_F\ \sigma} T\,B}$$

Figure 5: Structure-Annotated Type System for E.

$$\frac{\vdash s \;:\; A \xrightarrow{\sigma} B}{\vdash \mathsf{fun}\ s \;:\; T\,A \xrightarrow{\mathrm{FUN}\ \sigma} T\,B}
\qquad
\frac{\vdash s_1 \;:\; F\,B \xrightarrow{\sigma_1} B \qquad \vdash s_2 \;:\; A \xrightarrow{\sigma_2} F\,A \qquad G = \mathsf{base}\ F}{\vdash \mathsf{dc}_{n,F}\ \ s_1\ s_2 \;:\; T\,A \xrightarrow{\mathrm{DC}_{n,G}\ \sigma_1\ \sigma_2} T\,B}$$

$$\frac{n : \mathbb{N} \qquad \vdash p \;:\; T\,A \xrightarrow{\sigma} T\,B}{\vdash \mathsf{farm}\ n\ p \;:\; T\,A \xrightarrow{\mathrm{FARM}_n\ \sigma} T\,B}
\qquad
\frac{\vdash p_1 \;:\; T\,A \xrightarrow{\sigma_1} T\,B \qquad \vdash p_2 \;:\; T\,B \xrightarrow{\sigma_2} T\,C}{\vdash p_1 \parallel p_2 \;:\; T\,A \xrightarrow{\sigma_1 \parallel \sigma_2} T\,C}
\qquad
\frac{\vdash p \;:\; T\,A \xrightarrow{\sigma} T\,(A+B)}{\vdash \mathsf{fb}\ \ p \;:\; T\,A \xrightarrow{\mathrm{FB}\ \sigma} T\,B}$$

- $\sigma \sim A \to B$ is an alternative notation for $A \overset{\sigma}{\mapsto} B$

$$\frac{\vdash e \;:\; A \xrightarrow{\sigma_1} B \qquad \sigma_1 \cong \sigma_2}{\vdash e \;:\; A \xmapsto{\sigma_2} B}$$