Synthesizing MPI Implementations from Functional Data-Parallel Programs

Inferring data distributions using types

Tristan Aubrey-Jones
Bernd Fischer

UNIVERSITY OF Southampton

UNIVERSITEIT·STELLENBOSCH·UNIVERSITY
jou kennisvennoot • your knowledge partner

People need to program distributed memory architectures:

University of Southampton

GPUs

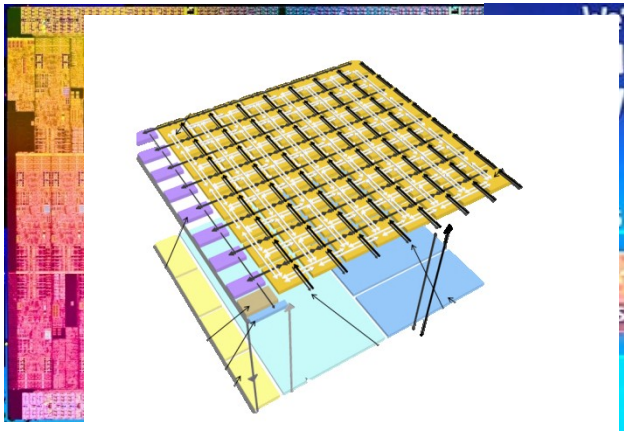
Graphics/GPGPU
CUDA/OpenCL

Memory levels:
thread-local,
block-shared,
device-global.

Server farms/compute clusters

Big Data/HPC; MapReduce/HPF; Ethernet/Infiniband

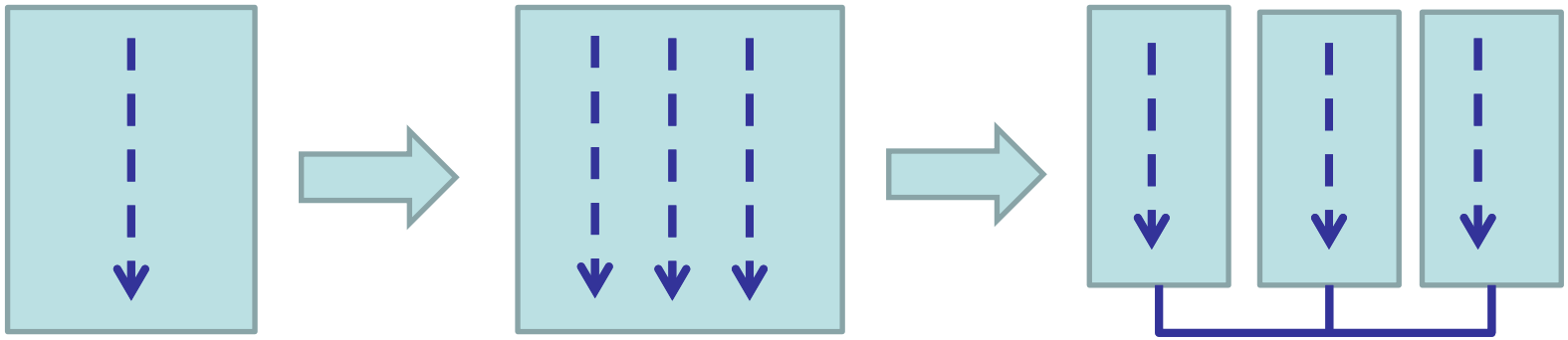
Future many-core architectures?



Our Aim

University of Southampton

To automatically generate distributed-memory cluster implementations (C++/MPI).



Single threaded | Parallel shared-memory | Distributed memory

Many distributions possible

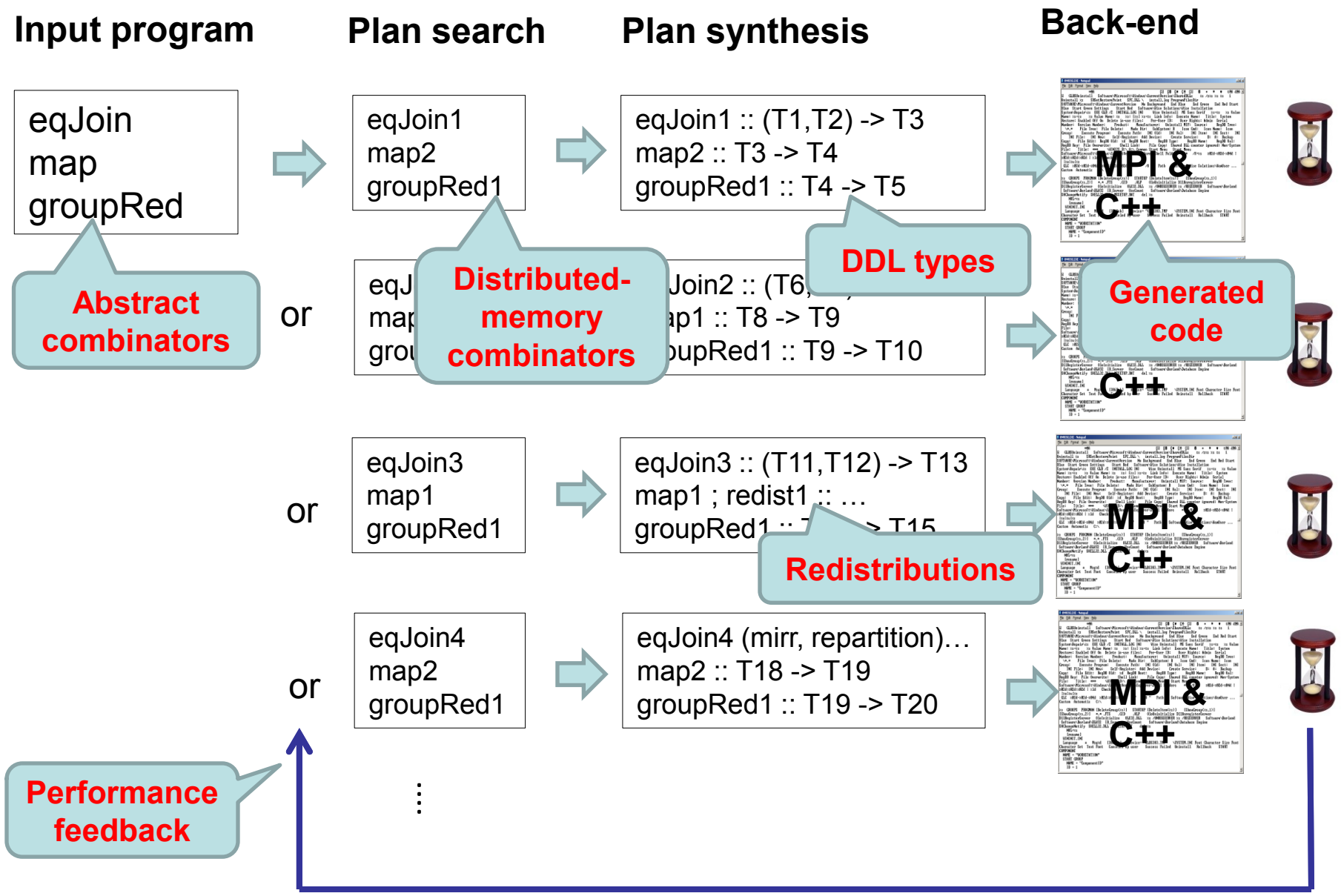
We want this **not just for arrays** (i.e., other collection types as well as disk backed collections).

Approach

University of Southampton

- We define Flocc, a **data-parallel DSL,** where parallelism is expressed via skeletons or **combinator functions** (HOFs).
- We use **distributed data layout (DDL) types** to carry data distribution information, and **type inference** to drive code generation.
- We develop a **code generator** that **searches** for well performing implementations, and generates **C++/MPI implementations**.

Approach


Input program
Plan search
Plan synthesis
Back-end
eqJoin
map
groupRed
Abstract combinators
or
eqJoin1
map2
groupRed1
Distributed-memory combinators
eqJoin1 :: (T1,T2) -> T3
map2 :: T3 -> T4
groupRed1 :: T4 -> T5
DDL types
MPI & C++
Generated code
C++
or
eqJoin3
map1
groupRed1
eqJoin3 :: (T11,T12) -> T13
map1 ; redist1 :: …
Redistributions
MPI & C++
or
eqJoin4
map2
groupRed1
eqJoin4 (mirr, repartition)…
map2 :: T18 -> T19
groupRed1 :: T19 -> T20
MPI & C++
Performance feedback

What's new?

University of Southampton

Last++ time:

- Functional DSL with data-parallel combinators
- Data distributions for distributed-memory implementations using dependent types
- Distributions for maps, arrays, and lists
- Synthesis of data distributions using type inference algorithm

Since then:

- MPI/C++ code generation
- Performance-feedback-based data distribution search
- Automatic redistribution insertion
- Local data layouts
- Arrays with ghosting
- Type inference with E-unification
- *(Thesis submitted)*

What we presented last time.

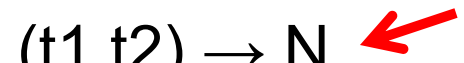
RE-CAP

A Distributed Map type

The **DMap** type extends the basic **Map k v** type to symbolically describe how the Map should be distributed on the cluster

$$dt ::= \ldots \mid \texttt{DMap}\ t_1\ t_2\ f\ d_1\ d_2 \mid \ldots$$

- Key and value types **t1** and **t2**
- Partition function **f**: (t1,t2) → N



  - takes key-value pairs to node coordinates
- Partition dimension identifier **d1**
  - specifies which nodes the coordinates map onto
- Mirror dimension identifier **d2**
  - specifies which nodes to mirror the partitions over

Also works for **Arrays (DArr)** and **Lists (DList)**

Distribution types for group reduces

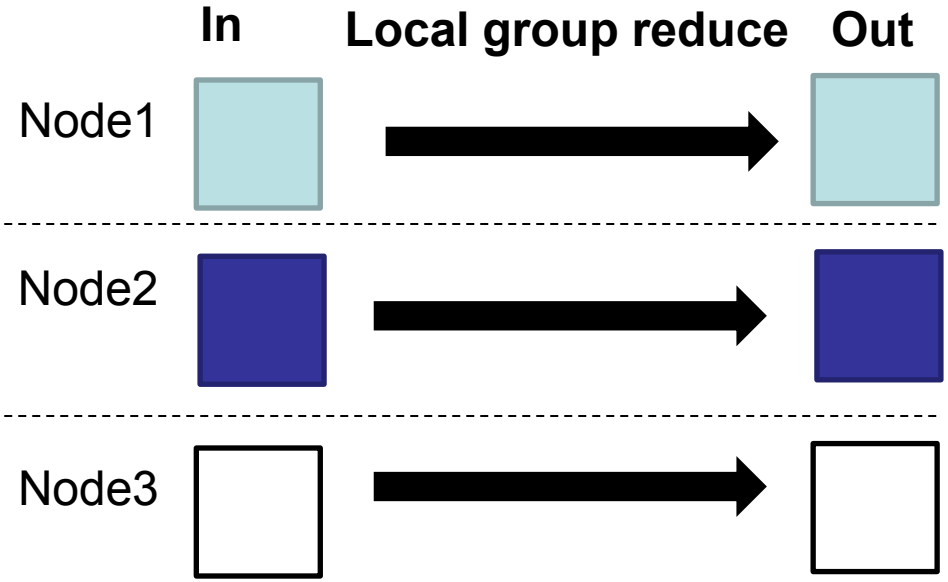
groupReduce2 : ∀k, k', v, v', d, m, Π(f,_,_,_) :
((k,v) → k', (k,v) → v', (v',v') → v', DMap k v f d m)
→ DMap k' v' fst d m

- Π binds concrete term in AST
- creates a reference to argument's AST when instantiated
- must match concrete reference when unifying (*rigid*)
- used to specify that a collection is distributed using a specific key projection function

Distribution types for group reduces

groupReduce2 : ∀k, k', v, v', d, m, Π(f,_,_,_) :
((k,v) → k', (k,v) → v', (v',v') → v', DMap k v f d m)
→ DMap k' v' fst d m

- Input must be partitioned using the groupReduce's key projection function f
- Result keys are already co-located



No inter-node communication necessary
(but constrains input distribution)

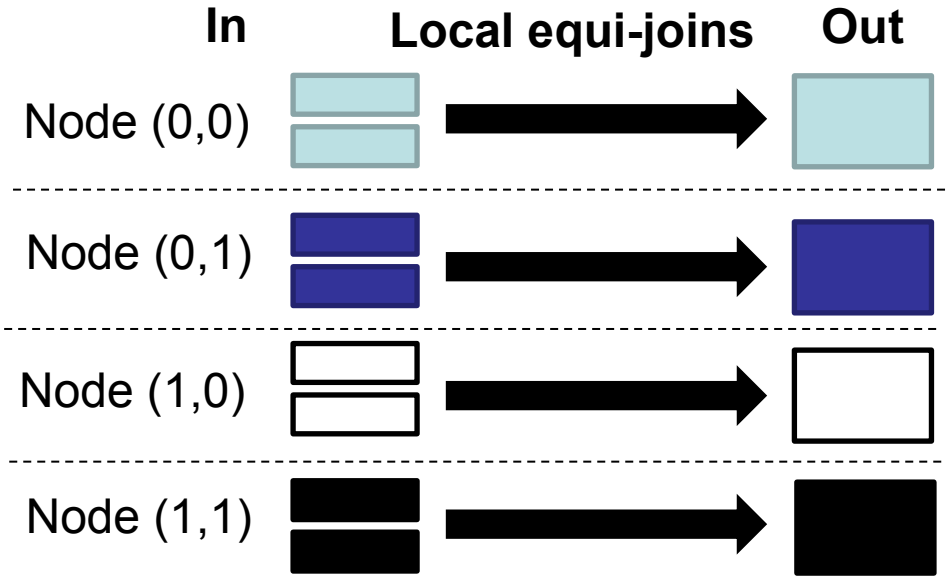
Distribution types for joins

eqJoin3 : ∀f, k', k1, k2, v1, v2, d1, d2, m,
((k1,v1) → k', (k2,v2) → k',
DMap k1 v1 lftFun(f) d1 (d2,m),
DMap k2 v2 rhtFun(f) d2, (d1,m))
→ DMap (k1,k2) (v1,v2) f (d1,d2) m

	0	1
0	A_0 B_0	A_1 B_0
1	A_0 B_1	A_1 B_1

d1

Left and right input partitioned and mirrored on orthogonal dims



- **No inter-node communication**
- **Output can be partitioned by any f**
- **Can be more efficient than mirroring whole input**

Deriving distribution plans

Hidden from user

- Different **distributed implementations** of each combinator
- Enumerate different choices of combinator implementations
- Each combinator implementation has a **DDL type**
- Use **type inference** to check if a set of choices is sound and infer data distributions
- **Backend templates** for each combinator implementation

eqJoin1 / eqJoin2 / eqJoin3

```
let R = eqJoin (\((ai,aj),_) -> aj,
                \((bi,bj),_) -> bi,
                A, B) in
groupReduce (\(((ai,aj),(bi,bj)),_) -> (ai,bj),
             \(_,(av,bv)) -> mul (av,bv), add, R)
```

groupReduce1 / groupReduce 2

Distributed matrix multiplication - #1

University of Southampton

```
let R = eqJoin (\((ai,aj),_) -> aj,
                \((bi,bj),_) -> bi,
                A, B) in
groupReduce (\(((ai,aj),(bi,bj)),_) -> (ai,bj),
             \(_,(av,bv)) -> mul (av,bv), add, R)
```

A : DMap (Int,Int) Float \((ai,aj),_) → ai d1 (d2,m)

B : DMap (Int,Int) Float \((bi,bj),_) → bj d2 (d1,m)

R : DMap ((Int,Int),(Int,Int)) (Float,Float)

\(((ai,aj),(bi,bj)),_) → (ai,bj) (d1,d2) m

C :

eqJoin3 : ∀f, k', k1, k2, v1, v2, d1, d2, m,
((k1,v1) → k', (k2,v2) → k',
DMap k1 v1 lftFun(f) d1 (d2,m),
DMap k2 v2 rhtFun(f) d2, (d1,m))
→ DMap (k1,k2) (v1,v2) f (d1,d2) m

d2)

M

omputation.

Distributed matrix multiplication - #1

University of Southampton

```
let R = eqJoin (\((ai,aj),_) -> aj,
                \((bi,bj),_) -> bi,
                A, B) in
groupReduce (\(((ai,aj),(bi,bj)),_) -> (ai,bj),
             \(_,(av,bv)) -> mul (av,bv), add, R)
```

A : DMap (Int,Int) Float \((ai,aj),_) → ai d1 (d2,m)

B : DMap (Int,Int) Float \((bi,bj),_) → bj d2 (d1,m)

R : DMap ((Int,Int),(Int,Int)) (Float,Float)

\(((ai,aj),(bi,bj)),_) → (ai,bj) (d1,d2) m

C : DMap (Int,Int) Float fst (d1,d2) m

groupReduce2 : ∀k, k', v, v', d, m, Π(f,_,_,_) :
((k,v) → k', (k,v) → v', (v',v') → v', DMap k v f d m)
→ DMap k' v' fst d m

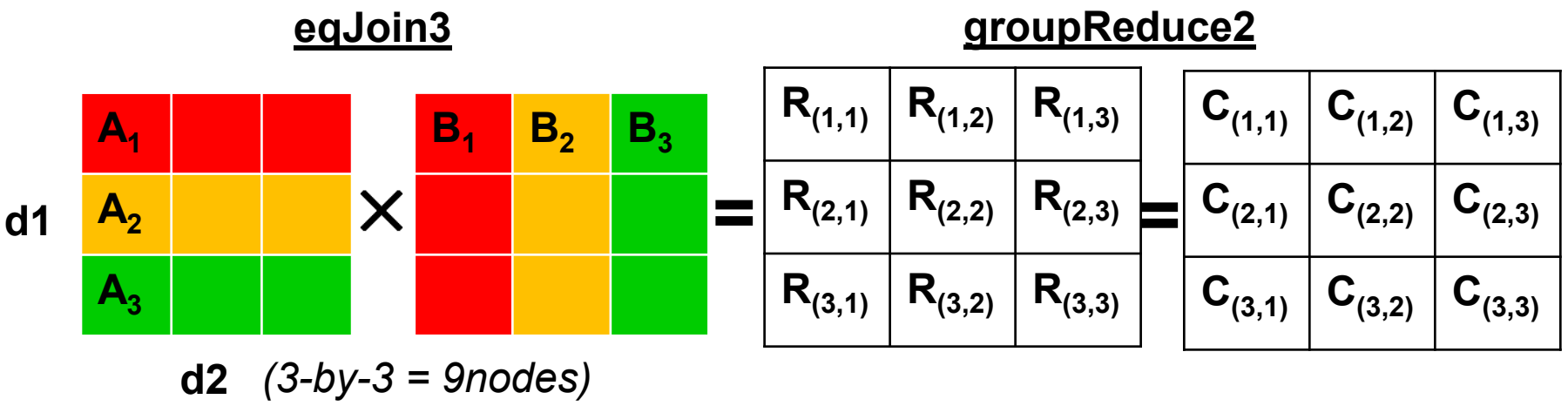
Distributed matrix multiplication - #1

University of Southampton

- **groupReduce2** – in part by key Πf, out part by fst (out keys)
- **eqJoin3** – left part by lftFun(f), right rhtFun(f), out by any f

A: Partitioned by row along d1, mirrored along d2

B: Partitioned by column along d2, mirrored along d1



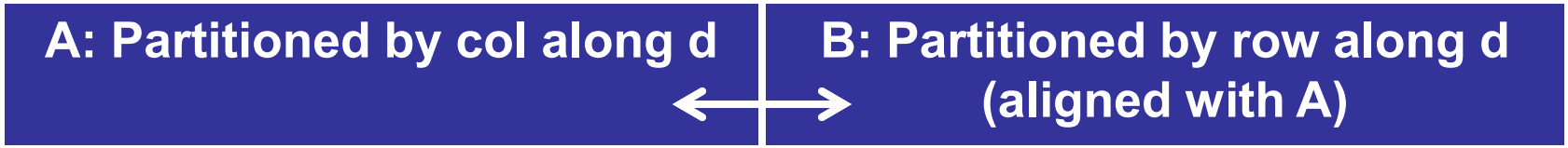
C: Partitioned by (row, column) along (d1, d2)

A common solution.

Distributed matrix multiplication - #2

University of Southampton

- **eqJoin1** – left part by Πf1, right by Πf2, aligned along d
- **groupReduce1** – in part by any f, out part by fst (out keys)




```
A : DMap (Int,Int) Float  \((ai,aj),_)->aj  d m
B : DMap (Int,Int) Float  \((bi,bj),_)->bi  d m
R : DMap ((Int,Int),(Int,Int)) (Float,Float)
      ((\((ai,aj),_) ->aj) · lft) d m
C : DMap (Int,Int) Float fst d m
```

C: Partitioned by (row, column) along d

Must exchange R during groupReduce1.

What's new since last time.

RECENT WORK

Since last time…

University of Southampton

- Local data layouts
- Distributed arrays with ghosting
- Automatic redistribution insertion
- E-unification in type inference
- MPI/C++ code generation
- Performance-feedback-based data distribution search
- Proof of concept (~25k Haskell loc)

Local data layouts

University of Southampton

- Extra DDL type parameters for local layouts
  - Choice of data structure (or storage mode)
    - ▷ Sorted std::vector, hash-map, tree-map, value-stream etc…
  - Order of elements or key indexed by
    - ▷ Local layout function similar to partition function

```
groupReduce2 :: Π(f,_,_,_) : ((i,v)->j, (i,v)->w, (w,w)->w,
  DMap ⟨...⟩ Stm f) -> DMap ⟨...⟩ Stm fst

union :: (DMap ⟨...⟩ Stm fst, DMap ⟨...⟩ Iter fst) -> DMap ⟨...⟩ Stm fst

diff1ii :: (DMap ⟨...⟩ Stm f, DMap ⟨...⟩ Hash fst) -> DMap ⟨...⟩ Stm f

diff1iii :: (DMap ⟨...⟩ Stm f, DMap ⟨...⟩ Tree fst) -> DMap ⟨...⟩ Stm f

diff1iv :: (DMap ⟨...⟩ Stm f, DMap ⟨...⟩ Vec fst) -> DMap ⟨...⟩ Stm f
```

Extended array distributions

University of Southampton

- Supports
  - index offsets, axis-reversal (all HPF alignments)
  - cyclic, blocked, and block-cyclic distributions (all HPF dists)
  - ghosted regions/fringes
- Index transformer functions in DDL types
  - Take and return tuples of integer array indices
    - ▷ Block-sizes
    - ▷ Index directions
    - ▷ Index offsets
    - ▷ Index ghosted fringe sizes (left and right)
- Distribution for Jacobi 1D
  - DArr … bs dir id id (+1) (+1) -> DArr … bs dir id id id
- Relies on E-unification (later…)

Automatic redistribution insertion

University of Southampton

- Data re-distribution and re-layout functions are type casts.
- For invalid Flocc plans (i.e., that don't type check) insert just enough redistributions (or re-layouts) to make type check.
- Means can synthesize a valid plan for any choice of combinator implementations.
- Finds implementations that benefit from redistributing data so more efficient combinator implementations can be used.

```
mirrMap :: DMap k v f d m -> DMap k v  f d (m,m')
repartMap :: DMap k v f1 d1 m -> DMap k v f2 d2 m

saveVecMap    :: DMap ⟨...⟩ Stm f  -> DMap ⟨...⟩ Vec f
sortVecMap    :: DMap ⟨...⟩ Vec f  -> DMap ⟨...⟩ Vec g
readVecMap    :: DMap ⟨...⟩ Vec f  -> DMap ⟨...⟩ Iter f
readIterMap   :: DMap ⟨...⟩ Iter f -> DMap ⟨...⟩ Stm f
```

E-unification

- Adding equational theories for projection, permutation, and index transformation functions to DDL type inference
- Use E-prover and “question” conjectures to return values for existentially qualified variables
- Allows improved array distributions and more flexible DDL types to be supported
- (Not integrated with current prototype)

E-unification: projection functions

University of Southampton

$$
\begin{array}{lcl}
\beta \cdot \alpha & \stackrel{def}{=} & \backslash \mathtt{x} \to (\beta(\alpha\ \mathtt{x})) \\
\alpha \otimes \beta & \stackrel{def}{=} & \backslash(\mathtt{x},\mathtt{y}) \to (\alpha\ \mathtt{x},\ \beta\ \mathtt{y}) \\
\mathrm{id} & \stackrel{def}{=} & \backslash \mathtt{x} \to \mathtt{x} \\
\Delta & \stackrel{def}{=} & \backslash \mathtt{x} \to (\mathtt{x},\mathtt{x}) \\
\Pi_1 & \stackrel{def}{=} & \backslash(\mathtt{x},\mathtt{y}) \to \mathtt{x} \\
\Pi_2 & \stackrel{def}{=} & \backslash(\mathtt{x},\mathtt{y}) \to \mathtt{y}
\end{array}
$$

$$f \cdot \mathrm{id} = f$$

$$\mathrm{id} \cdot f = f$$

$$\Pi_1 \cdot \Delta = \mathrm{id}$$

$$\Pi_2 \cdot \Delta = \mathrm{id}$$

$$(\Pi_1 \otimes \Pi_2) \cdot \Delta = \mathrm{id} \otimes \mathrm{id}$$

$$(\Pi_2 \otimes \Pi_1) \cdot \Delta \cdot (\Pi_2 \otimes \Pi_1) \cdot \Delta = \mathrm{id} \otimes \mathrm{id}$$

$$(f_1 \otimes f_2) \cdot (\Pi_2 \otimes \Pi_1) \cdot \Delta = (\Pi_2 \otimes \Pi_1) \cdot \Delta \cdot (f_2 \otimes f_1)$$

$$\Pi_1 \cdot (f_1 \otimes f_2) = f_1 \cdot \Pi_1$$

$$\Pi_2 \cdot (f_1 \otimes f_2) = f_2 \cdot \Pi_2$$

$$\Pi_1 \cdot (f_1 \otimes f_2) \cdot \Delta = f_1$$

$$\Pi_2 \cdot (f_1 \otimes f_2) \cdot \Delta = f_2$$

$$\Delta \cdot f = (f \otimes f) \cdot \Delta$$

$$(f_1 \cdot f_2) \cdot f_3 = f_1 \cdot (f_2 \cdot f_3)$$

$$(f_1 \otimes f_2) \cdot (f_3 \otimes f_4) = (f_1 \cdot f_3) \otimes (f_2 \cdot f_4)$$

E-unification: indexing functions

$$
\begin{array}{lcl}
\mathbf{0} & \stackrel{def}{=} & (0,\ \ldots,\ 0) \\
\mathbf{1} & \stackrel{def}{=} & (1,\ \ldots,\ 1) \\
i^{-1} & \stackrel{def}{=} & \texttt{diviv}\ (\mathbf{1}, i) \\
+(i) & \stackrel{def}{=} & \backslash \texttt{x} \rightarrow \texttt{addiv}\ (\texttt{x}, i) \\
-(i) & \stackrel{def}{=} & \backslash \texttt{x} \rightarrow \texttt{subiv}\ (\texttt{x}, i) \\
\times(i) & \stackrel{def}{=} & \backslash \texttt{x} \rightarrow \texttt{muliv}\ (\texttt{x}, i)
\end{array}
$$

$$
\begin{aligned}
i^{-1^{-1}} &= i \\
\mathbf{1}^{-1} &= \mathbf{1} \\
+(i) \cdot -(i) &= \mathrm{id} \\
-(i) \cdot +(i) &= \mathrm{id} \\
\times(i) \cdot \times(i^{-1}) &= \mathrm{id} \\
\times(i^{-1}) \cdot \times(i) &= \mathrm{id} \\
g \cdot \mathrm{id} &= g \\
\mathrm{id} \cdot g &= g \\
(g_1 \cdot g_2) \cdot g_3 &= g_1 \cdot (g_2 \cdot g_3) \\
(g_1 \otimes g_2) \cdot (g_3 \otimes g_4) &= (g_1 \cdot g_3) \otimes (g_2 \cdot g_4)
\end{aligned}
$$

E-unification: permutation functions

$$\begin{array}{lcl} \alpha \odot \beta & \stackrel{def}{=} & \backslash \mathtt{x} \rightarrow (\alpha\ \mathtt{x},\ \beta\ \mathtt{x}) \\ \mathrm{null} & \stackrel{def}{=} & \backslash \_ \rightarrow () \end{array}$$

$$h \cdot \mathrm{id} = h$$

$$\mathrm{id} \cdot h = h$$

$$h \odot \mathrm{null} = h$$

$$\mathrm{null} \odot h = h$$

$$(h_1 \cdot h_2) \cdot h_3 = h_1 \cdot (h_2 \cdot h_3)$$

$$(h_1 \odot h_2) \odot h_3 = h_1 \odot (h_2 \odot h_3)$$

$$(h_1 \odot h_2) \cdot h_3 = ((h_1 \cdot h_3) \odot (h_2 \cdot h_3))$$

Code generation

University of Southampton

- Generates C++ and MPI from plans (i.e., Flocc programs with concrete combinator implementations and inferred DDL types)
- Transforms to DFG and uses expression templates to generate code.
- Currently supports map- and list-based combinator templates.
- Uses “stream” local storage mode to splice multiple consumers into a producer’s loop body.

```
1  // BEGIN readList
2  int v14 = 0;
3
4  // BEGIN reduceList init
5  double v17(v1);
6  double v18;
7  // END reduceList init
8
9  std::vector<double >::iterator v15(v6->begin());
10 v14++;
11 std::vector<double >::iterator v12(v11->begin());
12 std::vector<double >::iterator v13(v11->end());
13
14 for (; v12 != v13 && v14 > 0; v12++) {
15   // BEGIN zip consume:
16   if (v15 < v6->end()) {
17   // BEGIN mapList consume:
18       double v16;
19       v16 = (*v12) * (*v15);
20
21       // BEGIN reduceList consume
22       v18 = v17 + v16;
23       v17 = v18;
24       // END reduceList consume
25
26   // END mapList consume
27   }
28   else if (v15 == v6->end()) v14--;
29   v15++;
30   // END zip consume
31 }
32
33 // BEGIN reduceList fin
34 v18 = v17;
35 cartComm.Allreduce(&v18, &v17, sizeof(double), MPI_PACKED, v103);
36 if (cartComm != cartComm) {
37   cartComm.Bcast(&v17, sizeof(double), MPI_PACKED, rootRank);
38 }
39 // END reduceList fin
40 // END readList
```

Code generation

University of Southampton

Performance comparable with hand-coded versions:

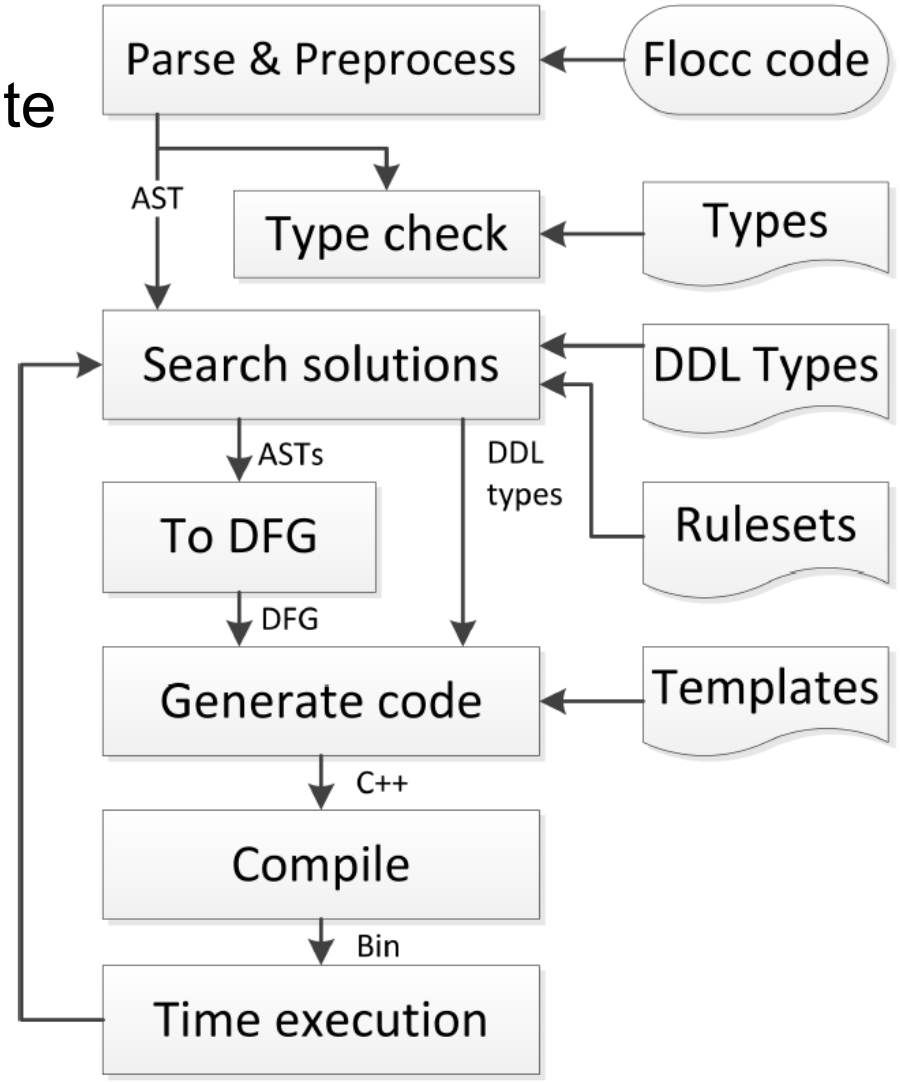
	PLINQ			Manual MPI		
Program	Speedup	Compiler	Data	Speedup	Compiler	Data
Dot product	4.96×	gcc -Ofast	2.2GB	0.99×	icc -O3	4.5GB
Simple linear regression	137×	gcc -Ofast	3GB	0.89×	icc -O3	3GB
				0.61×	gcc -O3	3GB
Standard deviation	98.6×	gcc -Ofast	3GB	0.88×	icc -O3	3GB
				1.00×	gcc -O3	3GB
Histogram	31.5×	gcc -Ofast	32MB	0.73×	icc -O3	8GB
Matrix multiply	342×	gcc -Ofast	1.4MB	6.14×	gcc -O3	140MB
				0.49×	icc -O3	140MB

- PLINQ comparisons run on quad-core (Intel Xeon W3520/2.67GHz) x64 desktop with 12GB RAM.
- C++/MPI comparisons run on Iridis3&4: a 3rd gen cluster with ~1000 Westmere compute nodes, each with two 6-core CPUs and 22GB RAM, over an InfiniBand network. Speedups compared to sequential, averaged over 1,2,3,4,8,9,16,32 nodes.

Performance-feedback-based search

University of Southampton

- Tried different search algorithms to explore candidate implementations
- For each candidate we **automatically insert redistributions** to make it type check
- We evaluate each candidate by code generating, compiling, and **running it on some test data**
- Generates C++ using MPI



Performance-feedback-based search

University of Southampton

- Tried 946 different combinations of search heuristics applied to 4 map-based example programs
- Heuristics composed of
  - Search algorithms
    - ▷ Genetic searches (e.g., with/without crossover)
    - ▷ Depth/first exhaustive
    - ▷ Greedy
  - Termination conditions
  - Runtime pruning
- Found
  - Genetic-searches successfully reduce search time
  - Fixed budget termination best
  - Fixed budget runtime pruning best
  - Need to enumerate different redistribution insertion variants

The Pros and Cons…

University of Southampton

Benefits

- Multiple distributed collections: **Maps, Arrays, Lists**...
- Generates distributed algorithms **fully automatically**
- **Performance feedback more accurate/flexible than cost metrics**
- **Finds algorithms including redistributions**
- **Synthesizes local layouts**
- Can support **in-memory and disk backed** collections (e.g. for Big Data)

Limitations

- Current implementation mainly has list and map combinator backend templates
- Current implementation's redistribution insertion algorithm is slow

Future work

University of Southampton

- Extend prototype implementation.
  - Array combinator backend templates
  - Faster redistribution insertion
- Integrate equational theories with implementations.
- Support more distributed memory architectures (GPUs).
- Retrofit into an existing functional language.
- Similar type inference for imperative languages?
- *(pass PhD viva)*

QUESTIONS?