Improving Implicit Parallelism

José Manuel Calderón Trilla & Colin Runciman

"I thought the 'lazy functional languages are great for implicit parallelism' thing died out some time ago."

–Ben Lippmeier (Haskell ML, 2005)

The takeaway

Static analysis ***alone*** is not enough to achieve implicit parallelism.

The takeaway

Static analysis ***alone*** is not enough to achieve implicit parallelism.

We use profile directed feedback ***in addition to*** well-known static analysis techniques to achieve better results.

Motivation

Motivation

- Dream the dream

Motivation

- Dream the dream
- Multi-cores are everywhere

Motivation

Motivation

- The parallelism is there* (Church-Rosser)

Motivation

- The parallelism is there* (Church-Rosser)
- We shouldn’t view performance as ‘all or nothing’. It should be cost/benefit.

Motivation

- The parallelism is there* (Church-Rosser)
- We shouldn't view performance as 'all or nothing'. It should be cost/benefit.

*Mostly

‘par’ annotations

‘par’ annotations

- Simple way to introduce parallelism

‘par’ annotations

- Simple way to introduce parallelism
- Cheap when using a sparking model (Clack and Peyton Jones 1986)

‘par’ annotations

- Simple way to introduce parallelism
- Cheap when using a sparking model (Clack and Peyton Jones 1986)
- Lends itself to use in Strategies (Trinder et al. 1998, Marlow et al. 2010)

‘par’ annotations

‘par’ annotations

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

‘par’ annotations

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = let x = fib (n-1)
            y = fib (n-2)
        in x `par` y `seq` x + y
```

‘par’ annotations

‘par’ annotations

- `par` also lends itself to ‘switching’

‘par’ annotations

- `par` also lends itself to ‘switching’

```
par :: a -> b -> b
```

Takeaway: revisited

Takeaway: revisited

- Use static analysis to place `par`s throughout the program, generously
- Use profiling data to determine which `par`s should be switched off

Defunctionalisation

Defunctionalisation

- Two purposes:

Defunctionalisation

- Two purposes:
  - Necessary for projection analysis (Hinze 1995)

Defunctionalisation

- Two purposes:
  - Necessary for projection analysis (Hinze 1995)
  - Specialises par-sites

Defunctionalisation

```
pMap :: (a -> b) -> [a] -> [b]
pMap f []     = []
pMap f (x:xs) = y `par` y : pMap f xs
  where
    y = f x
```

Defunctionalisation

```
pMap_g :: [a] -> [b]
pMap_g []     = []
pMap_g (x:xs) = y `par` y : pMap_g xs
  where
    y = g x
```

par placement

par placement

- We want safety

par placement

- We want safety
    - Only spark sub-expressions that are needed

par placement

- We want safety
  - Only spark sub-expressions that are needed
  - Projections for strictness analysis can help us determine which arguments are needed and how much is needed
    (Hinze 1995)

Projections

Projections

```
data Context = CVar String
             | CRec String
             | CBot
             | CProd [Context]
             | CSum [(String, Context)]
             | CMu String Context
             | CStr Context
             | CLaz Context
```

Projections $\approx$ Strategies

Projections ≈ Strategies

- Projections: describe how much of a structure is needed

Projections ≈ Strategies

- Projections: describe how much of a structure is needed
- Strategies: describe how much of a structure to evaluate (in parallel)

Projections ≈ Strategies

- Projections: describe how much of a structure is needed
- Strategies: describe how much of a structure to evaluate (in parallel)
- Similar to Burn's "Evaluation Transformers" (Burn 1991)

Projections ≈ Strategies

- Example: Analysis determines a list can be fully evaluated

Projections ≈ Strategies

- Example: Analysis determines a list can be fully evaluated

```
sList :: Strategy a -> [a] -> ()
sList s []      = ()
sList s (x:xs) = s x `par` sList s xs
```

1990’s Version

1990’s Version

- We’re done.

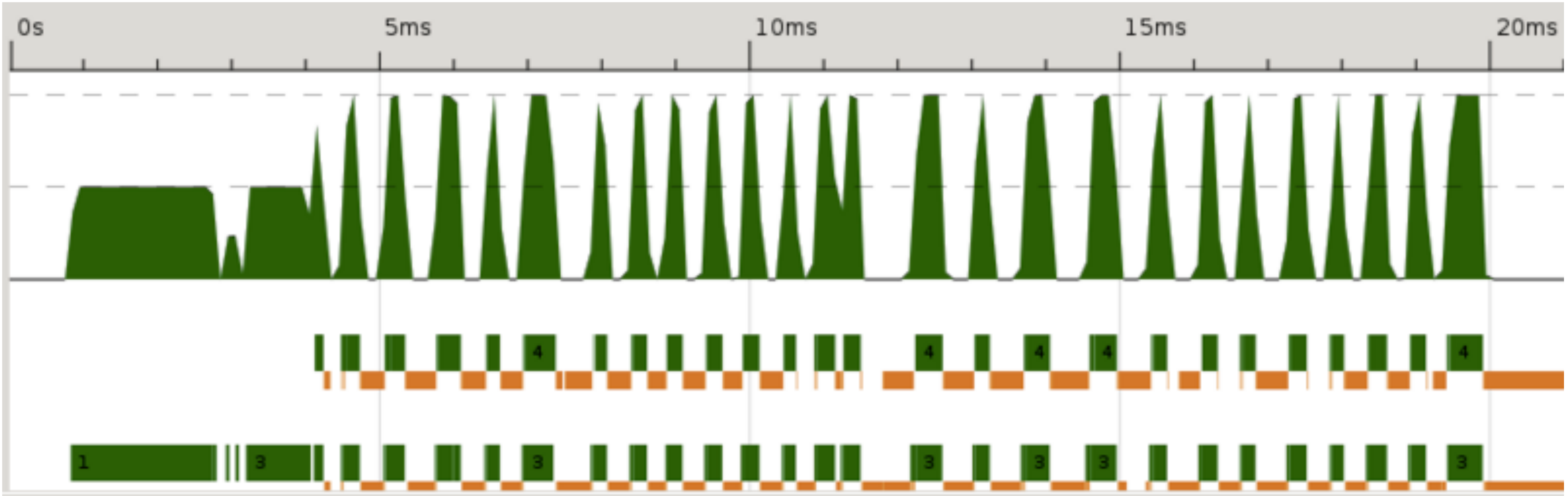
The remake

The remake

- Have the compiler do what programmers do: look at profiling data

The remake

- Have the compiler do what programmers do: look at profiling data



Par-site Health

Par-site Health

- Not all threads are equally productive

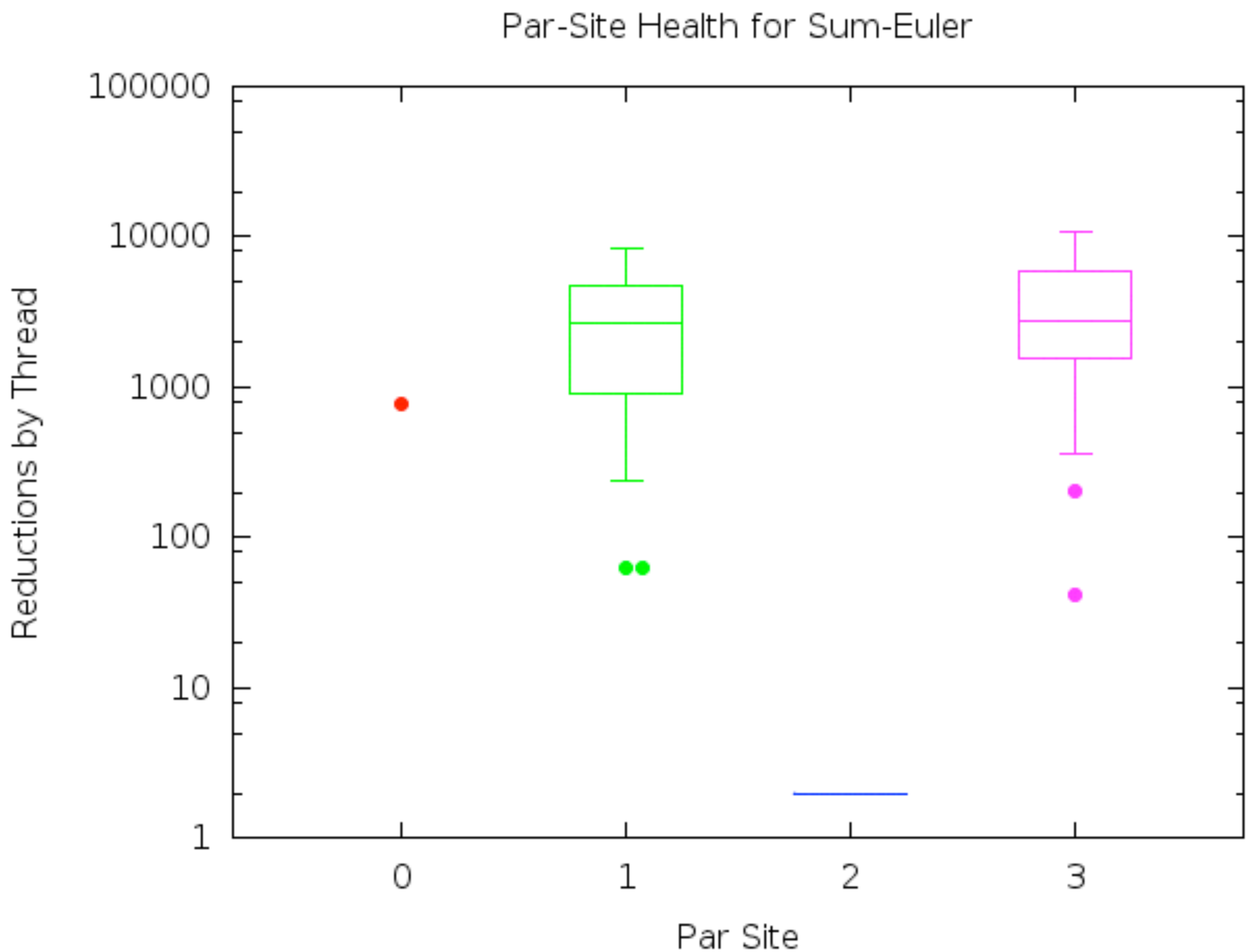
Par-site Health

- Not all threads are equally productive
- Each thread has an origin (par-site)

Par-site Health

- Not all threads are equally productive
- Each thread has an origin (par-site)
- Calculate the health of a par-site by looking at the productivity of the threads it sparked

Thread Health



Incorporate Feedback

Incorporate Feedback

- After calculating par-site health switch off the weakest `par`

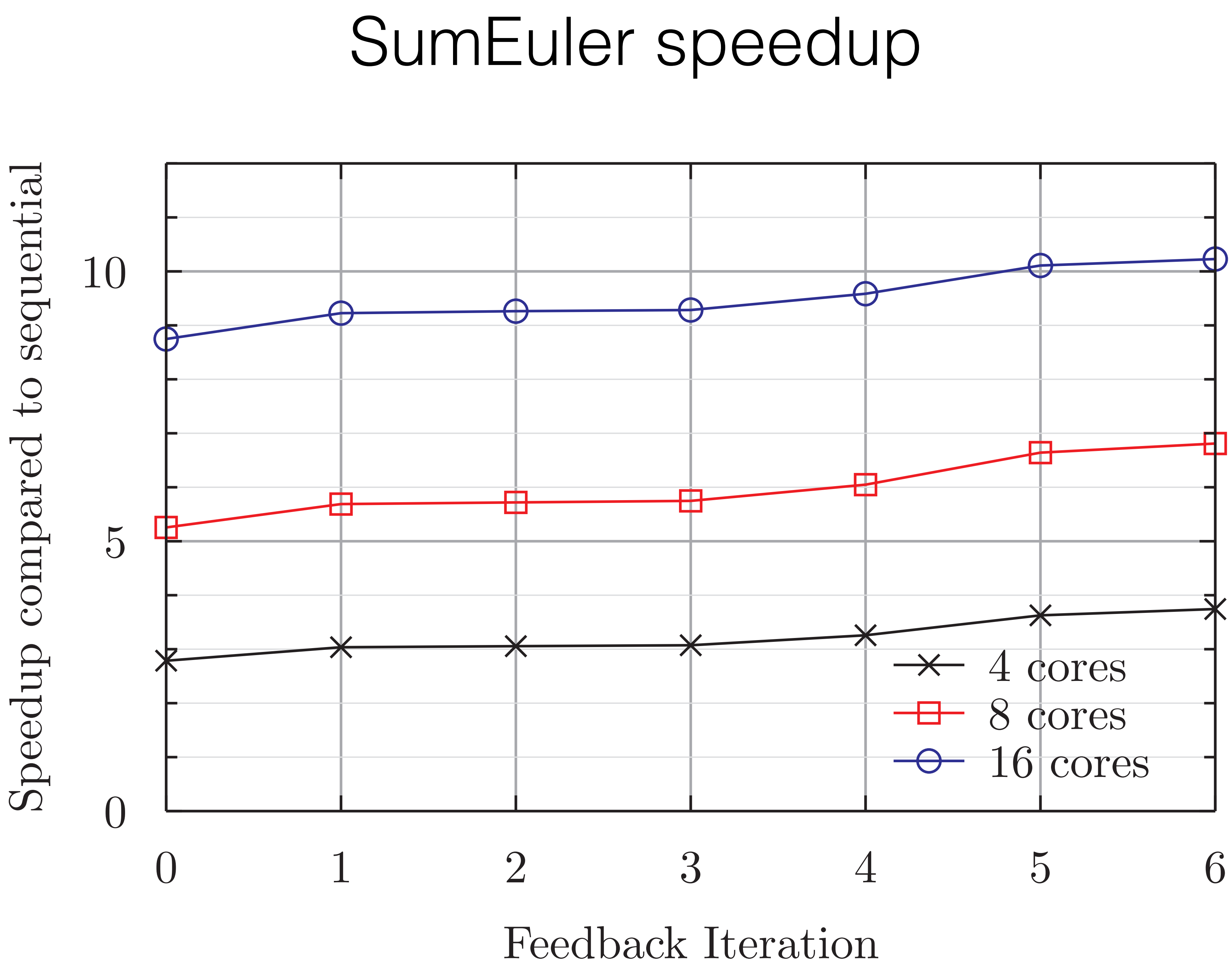
Incorporate Feedback

- After calculating par-site health switch off the weakest `par`
- Repeat until no more improvement to overall performance

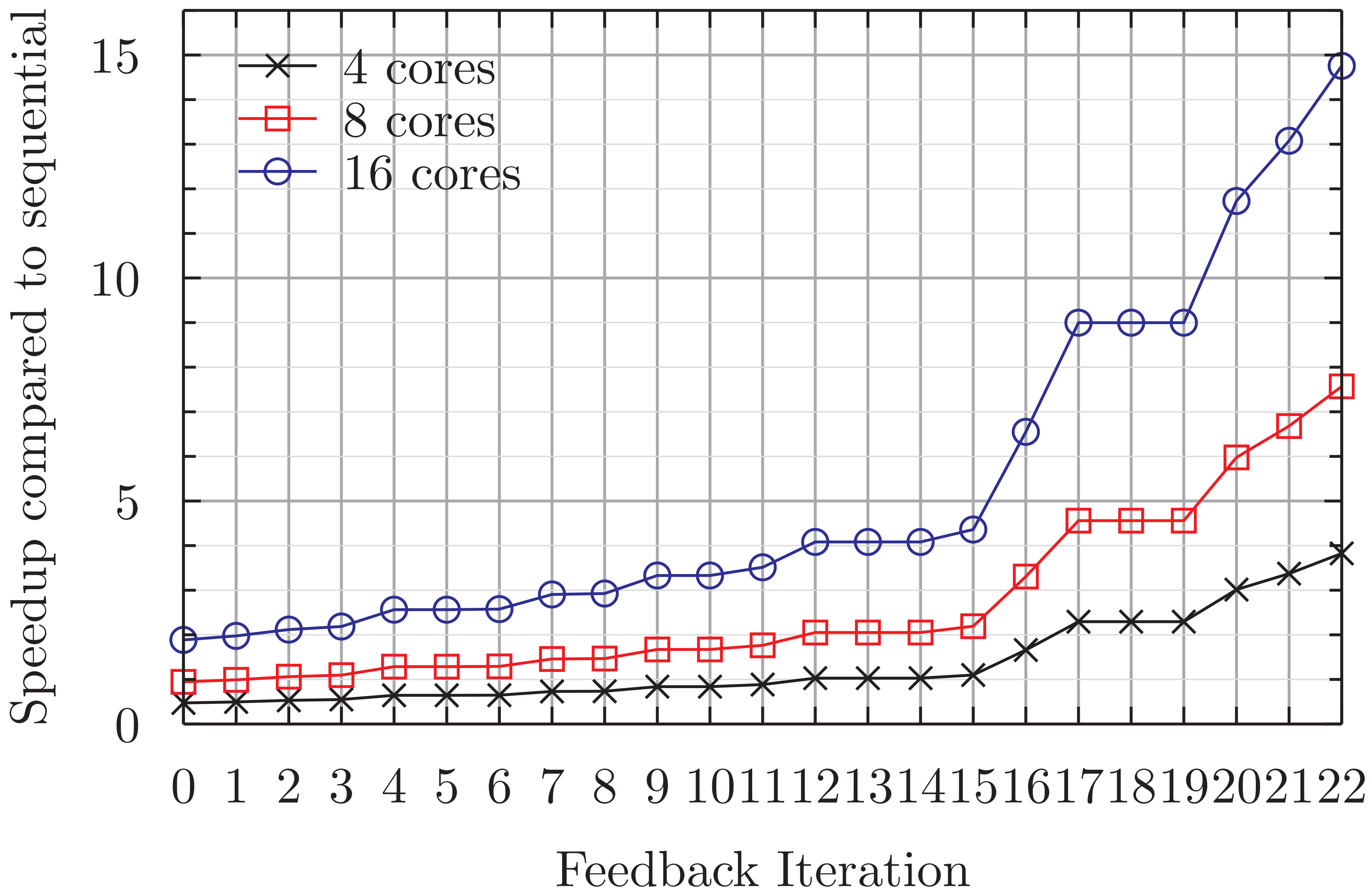
```
main  = let
    v_130 = let
      v_129 = fromto_D1 1 1000
      in
      par (fix mainLL_0 v_129) (mapDefeuler v_129))
    in
    par (fix mainLL_3 v_130) (sum v_130));

mainLL_2 v_0
  = seq v_0 Pack{0,0};

mainLL_1 v_1 v_2 = case v_2 of {
    <0> v_131 v_132 ->
      par (mainLL_2 v_131) (seq (v_1 v_132) Pack{0,0});
    <1> -> Pack{0,0}
    };

mainLL_0 v_3 = mainLL_1 v_3;

mainLL_5 v_4
  = seq v_4 Pack{0,0};

mainLL_4 v_5 v_6 = case v_6 of {
    <0> v_133 v_134 ->
      par (mainLL_5 v_133) (seq (v_5 v_134) Pack{0,0});
    <1> -> Pack{0,0}
    };

mainLL_3 v_7 = mainLL_4 v_7;

sum v_8 = case v_8 of {
    <1> -> 0;
    <0> v_135 v_136 -> let
        v_139 = sum v_136
        in
        (par (sumLL_0 v_139) ((v_135 + v_139)))
    };

sumLL_0 v_9 = seq v_9 Pack{0,0};

mapDefeuler v_10 = case v_10 of
    {
    <1> -> Pack{1,0};
    <0> v_140 v_141 ->
      Pack{0,2} (euler v_140) (mapDefeuler v_141)
    };

fromto_D1 v_11 v_12
  = ifte ((v_11 > v_12)) Pack{1,0}
       (Pack{0,2} v_11 (fromto_D1 ((v_11 + 1)) v_12));

fromto_D2 v_13 v_14
  = ifte ((v_13 > v_14)) Pack{1,0}
       (Pack{0,2} v_13 (fromto_D2 ((v_13 + 1)) v_14));

gcd v_30 v_31
  = ifte ((v_31 == 0)) v_30
       (ifte ((v_30 > v_31)) (gcd ((v_30 - v_31)) v_31)
                             (gcd v_30 ((v_31 - v_30))))
```

```
euler v_15 = let
    v_164 = filterDefrelPrime v_15 (fromto_D2 1 v_15)
    in
    (par (fix eulerLL_0 v_164) (length v_164));

eulerLL_1 v_16 v_17 = case v_17
  of {
    <0> v_168 v_169 ->
      seq (v_16 v_169) Pack{0,0};
    <1> -> Pack{0,0}
    };

eulerLL_0 v_18 = eulerLL_1 v_18;

ifte v_19 v_20 v_21 = case v_19
  of {
    <1> -> v_20;
    <0> -> v_21
    };

length v_22 = case v_22 of {
    <1> -> 0;
    <0> v_170 v_171 -> let
        v_174 = length v_171
        in
        (par (lengthLL_0 v_174) ((1 + v_174)))
    };

lengthLL_0 v_23
  = seq v_23 Pack{0,0};

filterDefrelPrime v_24 v_25
  = case v_25 of {
    <1> -> Pack{1,0};
    <0> v_175 v_176 -> let
        v_183 = relPrime v_24 v_175
        in
        par (filterDefrelPrimeLL_0 v_183)
            (ifte v_183 (Pack{0,2} v_175
                                      (filterDefrelPrime v_24 v_176))
                         (filterDefrelPrime v_24 v_176)))
    };

filterDefrelPrimeLL_0 v_26
  = case v_26 of {
    <1> -> Pack{0,0};
    <0> -> Pack{0,0}
    };

relPrime v_27 v_28 = let
    v_188 = gcd v_27 v_28
    in
    (par (relPrimeLL_0 v_188) ((v_188 == 1)));

relPrimeLL_0 v_29
  = seq v_29 Pack{0,0};
```

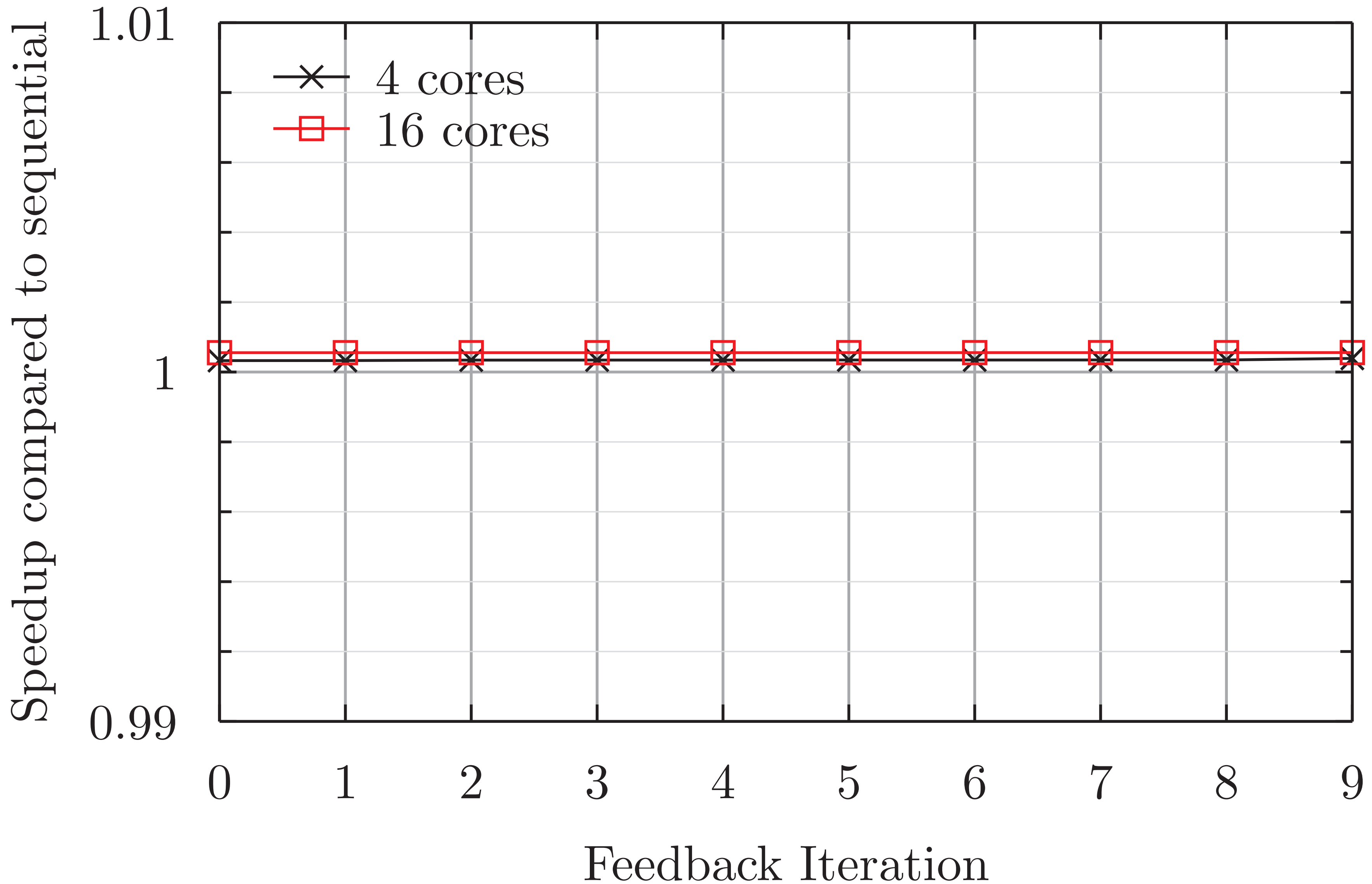
```
main  = let
    v_130 = let
      v_129 = fromto_D1 1 1000
      in
      par (fix mainLL_0 v_129) (mapDefeuler v_129))
    in
    par (fix mainLL_3 v_130) (sum v_130));

mainLL_2 v_0
  = seq v_0 Pack{0,0};

mainLL_1 v_1 v_2 = case v_2 of {
    <0> v_131 v_132 ->
      [redacted] (mainLL_2 v_131) (seq (v_1 v_132) Pack{0,0});
    <1> -> Pack{0,0}
    };

mainLL_0 v_3 = mainLL_1 v_3;

mainLL_5 v_4
  = seq v_4 Pack{0,0};

mainLL_4 v_5 v_6 = case v_6 of {
    <0> v_133 v_134 ->
      par (mainLL_5 v_133) (seq (v_5 v_134) Pack{0,0});
    <1> -> Pack{0,0}
    };

mainLL_3 v_7 = mainLL_4 v_7;

sum v_8 = case v_8 of {
    <1> -> 0;
    <0> v_135 v_136 -> let
        v_139 = sum v_136
        in
        [redacted] (sumLL_0 v_139) ((v_135 + v_139)))
    };

sumLL_0 v_9 = seq v_9 Pack{0,0};

mapDefeuler v_10 = case v_10 of
    {
    <1> -> Pack{1,0};
    <0> v_140 v_141 ->
      Pack{0,2} (euler v_140) (mapDefeuler v_141)
    };

fromto_D1 v_11 v_12
  = ifte ((v_11 > v_12)) Pack{1,0}
         (Pack{0,2} v_11 (fromto_D1 ((v_11 + 1)) v_12));

fromto_D2 v_13 v_14
  = ifte ((v_13 > v_14)) Pack{1,0}
         (Pack{0,2} v_13 (fromto_D2 ((v_13 + 1)) v_14));

gcd v_30 v_31
  = ifte ((v_31 == 0)) v_30
         (ifte ((v_30 > v_31)) (gcd ((v_30 - v_31)) v_31)
                               (gcd v_30 ((v_31 - v_30))))

euler v_15 = let
    v_164 = filterDefrelPrime v_15 (fromto_D2 1 v_15)
    in
    [redacted] (fix eulerLL_0 v_164) (length v_164));

eulerLL_1 v_16 v_17 = case v_17
  of {
    <0> v_168 v_169 ->
      seq (v_16 v_169) Pack{0,0};
    <1> -> Pack{0,0}
    };

eulerLL_0 v_18 = eulerLL_1 v_18;

ifte v_19 v_20 v_21 = case v_19
  of {
    <1> -> v_20;
    <0> -> v_21
    };

length v_22 = case v_22 of {
    <1> -> 0;
    <0> v_170 v_171 -> let
        v_174 = length v_171
        in
        [redacted] (lengthLL_0 v_174) ((1 + v_174)))
    };

lengthLL_0 v_23
  = seq v_23 Pack{0,0};

filterDefrelPrime v_24 v_25
  = case v_25 of {
    <1> -> Pack{1,0};
    <0> v_175 v_176 -> let
        v_183 = relPrime v_24 v_175
        in
        [redacted] (filterDefrelPrimeLL_0 v_183)
            (ifte v_183 (Pack{0,2} v_175
                                  (filterDefrelPrime v_24 v_176))
                             (filterDefrelPrime v_24 v_176)))
    };

filterDefrelPrimeLL_0 v_26
  = case v_26 of {
    <1> -> Pack{0,0};
    <0> -> Pack{0,0}
    };

relPrime v_27 v_28 = let
    v_188 = gcd v_27 v_28
    in
    [redacted] (relPrimeLL_0 v_188) ((v_188 == 1)));

relPrimeLL_0 v_29
  = seq v_29 Pack{0,0};
```


SumEuler speedup
Speedup compared to sequential
Feedback Iteration
0
5
10
0
1
2
3
4
5
6
4 cores
8 cores
16 cores

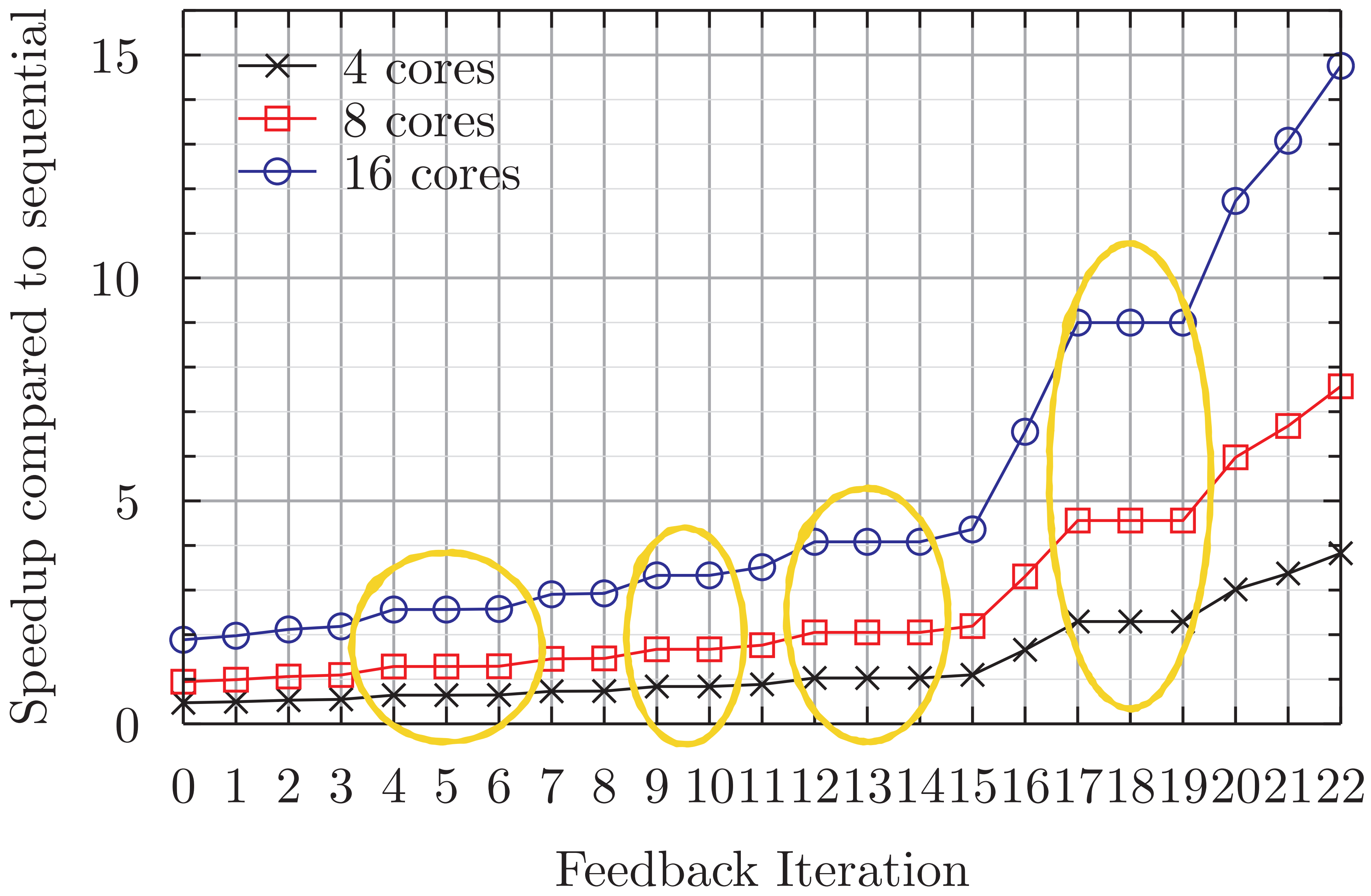
Queens2 speedup



Taut speedup



Queens2 speedup



Conclusion

Conclusion

- Early results promising

Conclusion

- Early results promising
- Will likely need other forms of specialisation

Conclusion

- Early results promising
- Will likely need other forms of specialisation
- Speculation may be necessary for more complex programs

Fin

Example Projections

```
Pairs:
CSum [("Pair", CProd [CProd []?, CProd []?])]
CSum [("Pair", CProd [CProd []!, CBot?])]


Lists:
CMu "L" (CSum ,[("Cons", CProd [(CVar "a")?
                                  ,(CRec "L")!])
               ("Nil", CProd [])]
```