Placement Strategies: Structured Skeleton Composition with Location Aware Remote Data

TFP 2020

Lukas Immanuel Schiller

AG Programmiersprachen und Parallelität
Prof. Dr. R. Loogen

14.02.2020

```
qsort [] = []
qsort (p:xs) = smaller' ++ [p] ++ larger'
    where
        smaller = filter (<p) xs
        larger  = filter (>=p) xs
        [smaller',larger'] = map qsort [smaller,larger]
```

Algorithmic Skeletons

Algorithmic Skeletons:

Algorithmic Skeletons

Algorithmic Skeletons:

- Implement common computation or communication pattern of parallel algorithms.
- typical patterns:
  - parMap, farm, workpool
  - map-reduce
  - divide and conquer
  - topology skeletons: pipeline, ring, torus, grid, hypercube

Algorithmic Skeletons

```
qsort [] = []
qsort (p:xs) = smaller' ++ [p] ++ larger'
    where
        smaller = filter (<p) xs
        larger  = filter (>=p) xs
        [smaller',larger'] = map qsort [smaller,larger]
```

Algorithmic Skeletons

```
qsort [] = []
qsort (p:xs) = smaller' ++ [p] ++ larger'
    where
        smaller = filter (<p) xs
        larger  = filter (>=p) xs
        [smaller',larger'] = map qsort [smaller,larger]
```

```
dc trivial solve split combine = rec_dc
  where
    rec_dc x = if trivial x then solve x
               else combine x (fmap rec_dc (split x))
```

Algorithmic Skeletons

```
qsort [] = []
qsort (p:xs) = smaller' ++ [p] ++ larger'
    where
        smaller = filter (<p) xs
        larger  = filter (>=p) xs
        [smaller',larger'] = map qsort [smaller,larger]
```

```
dc trivial solve split combine = rec_dc
  where
    rec_dc x = if trivial x then solve x
               else combine x (fmap rec_dc (split x))
```

```
qsort xs = dc trivial solve split combine xs where
    trivial = null
    solve = id
    split (p:xs) = (λ (l,g) → [l,g]) ∘ partition (< p) $ xs
    combine (p:_) (smaller':larger':_) = smaller' ++ [p] ++ larger'
```

Eden

Eden - “Semi-Explicit Control Parallelism”

Eden

Eden - “Semi-Explicit Control Parallelism”

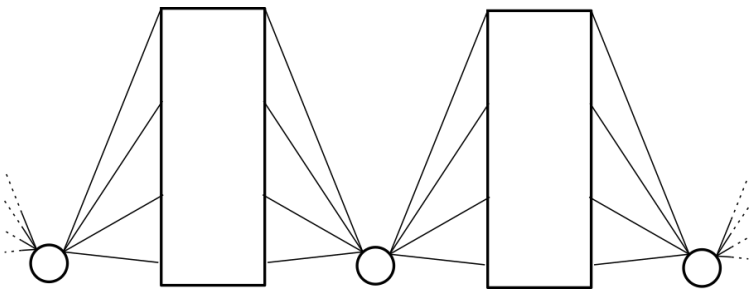
- parallel Haskell dialect with explicit parallel function application via parallel *processes* with implicit communication
- processes can be instantiated with explicit or semi-explicit placement
- distributed memory
- comes with skeleton library
- ...

Eden

```
class ParFunctor f where
    -- | parallel version of @'fmap'@.
    pmap :: (Trans a, Trans b) ⇒ (a → b) → f a → f b
```

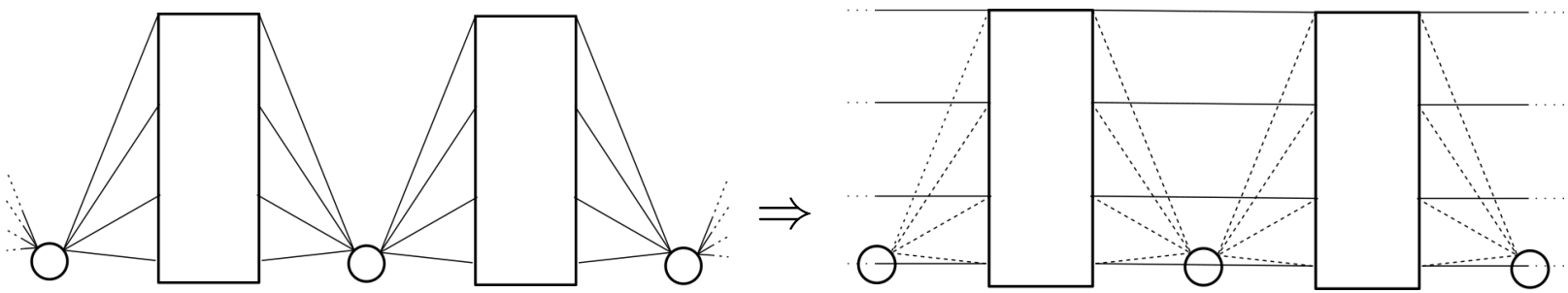
```
    pmapAt :: (Traversable t, Trans a, Trans b)
        ⇒ t Place      -- ^places for instantiation
        → (a → b)      -- ^worker function
        → f a          -- ^tasks
        → f b          -- ^results
```

Eden



skel1 ∘ skel0 $ inp

Remote Data



pmap g ∘ pmap f ⇒ pmap g' ∘ pmap f'

```
where
    g' = release ∘ g ∘ fetch
    f' = release ∘ f ∘ fetch
```

Remote Data

- Skeletons with “static” placement (cycle, block, . . .)
- Remote Data for skeleton composition
- granularity control (chunck, unshuffle, . . .)

Remote Data

- Skeletons with “static” placement (cycle, block, . . .)
- Remote Data for skeleton composition
- granularity control (chunck, unshuffle, . . .)

```
map f xs ⇝ pmap (release ∘ map f) (chunck n xs)
```

Remote Data

```
type RD a = ChanName (ChanName a)
```

```
data RD a = RD {place :: Place,
                rd :: ChanName (ChanName a) }
```

Remote Data

```
type RD a = ChanName (ChanName a)
```

```
data RD a = RD {place :: Place,
                rd :: ChanName (ChanName a) }
```

```
pmapAt places inp where
    places = fmap place inp
```

Co-location

```
rdmap :: (Trans a, Trans b, Functor t, ParFunctor t)
    ⇒ (a → b) -- ^map function
    → t (RD a) -- ^inputs
    → t (RD b) -- ^outputs
rdmap f = fmap (pmap f)
```

Introductory Example

```
res xs = merger map cElem xs
merger mapSkel cElem = s4 ∘ s4 ∘ s3 ∘ s2 ∘ s2 ∘ s1 where
    ac = mapSkel cElem
    s1 = perm1out ∘ ac ∘ perm1in
    s2 = perm2out ∘ ac ∘ perm2in
    s3 = perm3out ∘ ac ∘ perm3in
    s4 = perm4out ∘ ac ∘ perm4in
```

Introductory Example

```
res xs = merger pmap (releaseAll ∘ cElem ∘ fetchAll) xs
merger mapSkel cElem = s4 ∘ s4 ∘ s3 ∘ s2 ∘ s2 ∘ s1 where
    ac = mapSkel cElem
    s1 = perm1out ∘ ac ∘ perm1in
    s2 = perm2out ∘ ac ∘ perm2in
    s3 = perm3out ∘ ac ∘ perm3in
    s4 = perm4out ∘ ac ∘ perm4in
```

Introductory Example

```
res xs = merger pmap (releaseAll ∘ cElem ∘ fetchAll) xs
merger mapSkel cElem = s4 ∘ s4 ∘ s3 ∘ s2 ∘ s2 ∘ s1 where
    ac = mapSkel cElem
    s1 = perm1out ∘ ac ∘ perm1in
    s2 = perm2out ∘ ac ∘ perm2in
    s3 = perm3out ∘ ac ∘ perm3in
    s4 = perm4out ∘ ac ∘ perm4in
```

- places = [1,1,2,2] $\rightsquigarrow$ [1,3,2,4]

Introductory Example

```
res xs = merger pmap (releaseAll ∘ cElem ∘ fetchAll) xs
merger mapSkel cElem = s4 ∘ s4 ∘ s3 ∘ s2 ∘ s2 ∘ s1 where
    ac = mapSkel cElem
    s1 = perm1out ∘ ac ∘ perm1in
    s2 = perm2out ∘ ac ∘ perm2in
    s3 = perm3out ∘ ac ∘ perm3in
    s4 = perm4out ∘ ac ∘ perm4in
```

- places = [1,1,2,2] $\rightsquigarrow$ [1,3,2,4]
- places = [[1,2],[3,4],[1,2],[3,4]] $\rightsquigarrow$ [1,3,2,4]

Introductory Example

```
rdmapPStrat :: (ParFunctor f, Traversable t, Trans a, Trans b)
   ⇒ ((RD a1 → Place) → t Place → f a → t Place)  -- placement strategy
   → t Place                                     -- target places
   → (a → b)                                     -- map function
   → f a                                         -- input
   → f b                                         -- output
rdmapPStrat strat targets f xs = pmapAt places f xs where
    places = strat place targets xs
```

Introductory Example

```
bSortRDDC cElem targets d xss
 = rdPStratDC nextfreeidx targets trivial solve split combine xss where
   trivial = isDualton
   solve = rdapp (cElem Up)
   split _ = splitHalf
   combine targets _ = (bMergeRDDC cElem targets d) ∘ unSplit ∘ (<->) [id,reverse]

bMergeRDDC cElem targets d xss
 = rdPStratDC nextfreeidx targets trivial solve split combine xss where
   trivial = isDualton
   solve = (cElem d)
   split targets  = splitHalf ∘ shuffle ∘ rdmap' targets (cElem d) ∘ perfectShuffle
   combine _ _ = unSplit
   rdmap' = rdmapPStrat nextfreeidx
```

New Possibilities

scenario:

New Possibilities

scenario:

- heterogeneous architecture:
  for example beowulf cluster: 32 Computers with 8 Cores and 12 GB memory

New Possibilities

scenario:

- heterogeneous architecture:
  for example beowulf cluster: 32 Computers with 8 Cores and 12 GB memory

```
sorting_network ∘ presort ∘ workpool $ inp
```

New Possibilities

scenario:

- heterogeneous architecture:
  for example beowulf cluster: 32 Computers with 8 Cores and 12 GB memory

```
sorting_network ∘ presort ∘ workpool $ inp
```

- dynamic placement

New Possibilities

scenario:

- heterogeneous architecture:
  for example beowulf cluster: 32 Computers with 8 Cores and 12 GB memory

```
sorting_network ∘ presort ∘ workpool $ inp
```

- dynamic placement
- architecture awareness

New Possibilities

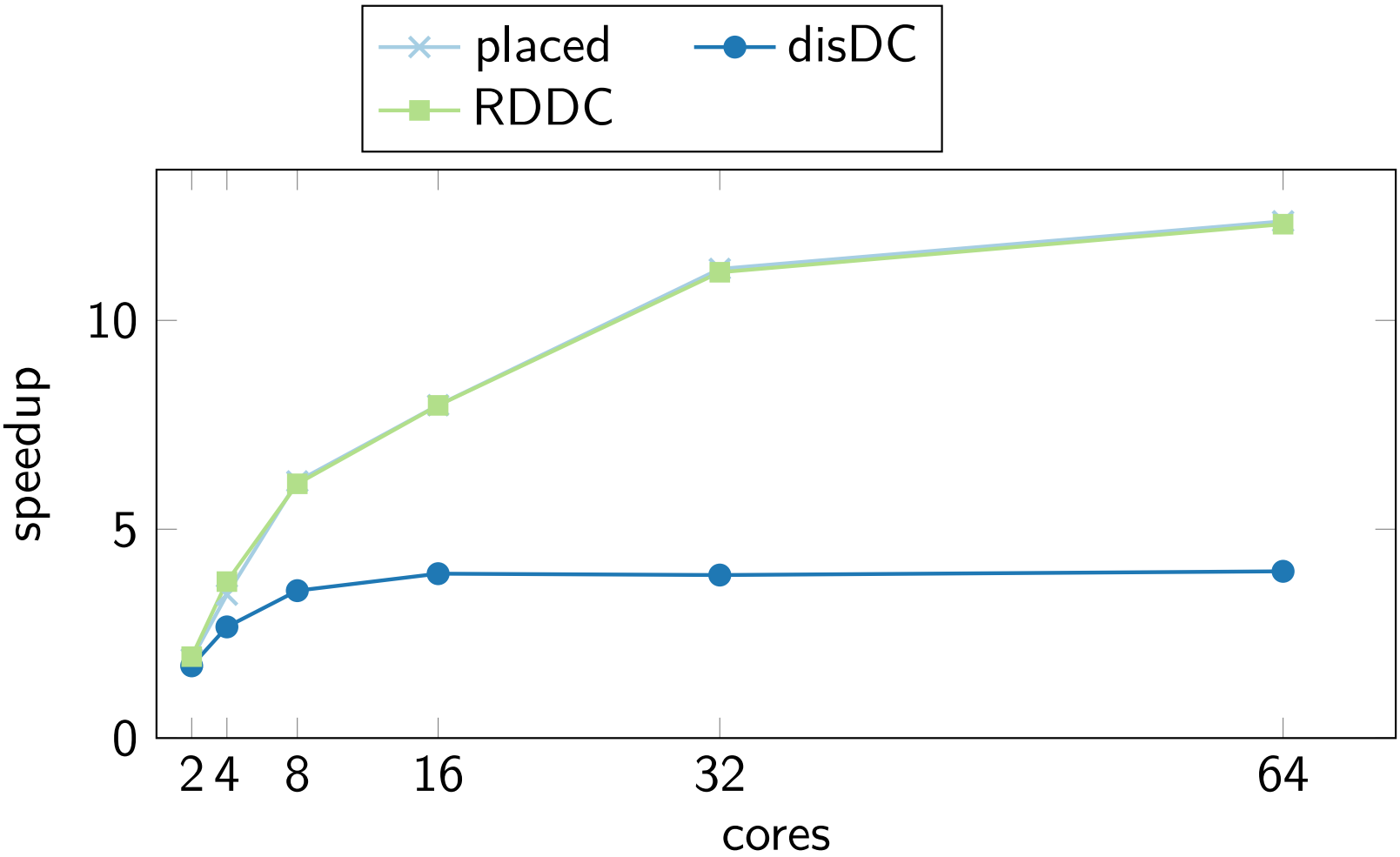
scenario:

- heterogeneous architecture:
  for example beowulf cluster: 32 Computers with 8 Cores and 12 GB memory

```
sorting_network ∘ presort ∘ workpool $ inp
```

- dynamic placement
- architecture awareness
- work-pulling vs. work-pushing

- Domain Maps
- scheduler
- alignment


placed
disDC
RDDC
speedup
0
5
10
2 4
8
16
32
64
cores

Conclusion

Conclusion

- explicit placement necessary/desired?

Conclusion

- explicit placement necessary/desired?
- overhead justified?

Conclusion

- explicit placement necessary/desired?
- overhead justified?
- much more flexible skeletons

Conclusion

- explicit placement necessary/desired?
- overhead justified?
- much more flexible skeletons
- easy change between control and data parallelism

Conclusion

- explicit placement necessary/desired?
- overhead justified?
- much more flexible skeletons
- easy change between control and data parallelism
- better nesting

Conclusion

- explicit placement necessary/desired?
- overhead justified?
- much more flexible skeletons
- easy change between control and data parallelism
- better nesting
- different strategies in the same program, work-pulling and work-pushing

Conclusion

- explicit placement necessary/desired?
- overhead justified?
- much more flexible skeletons
- easy change between control and data parallelism
- better nesting
- different strategies in the same program, work-pulling and work-pushing
- architecture awareness, cost functions

Conclusion

- explicit placement necessary/desired?
- overhead justified?
- much more flexible skeletons
- easy change between control and data parallelism
- better nesting
- different strategies in the same program, work-pulling and work-pushing
- architecture awareness, cost functions
- work in process:

Conclusion

- explicit placement necessary/desired?
- overhead justified?
- much more flexible skeletons
- easy change between control and data parallelism
- better nesting
- different strategies in the same program, work-pulling and work-pushing
- architecture awareness, cost functions
- work in process:
  - goal: complete flexible skeleton library

Conclusion

- explicit placement necessary/desired?
- overhead justified?
- much more flexible skeletons
- easy change between control and data parallelism
- better nesting
- different strategies in the same program, work-pulling and work-pushing
- architecture awareness, cost functions
- work in process:
  - goal: complete flexible skeleton library
  - alignment/semi-explicit placement

Conclusion

- explicit placement necessary/desired?
- overhead justified?
- much more flexible skeletons
- easy change between control and data parallelism
- better nesting
- different strategies in the same program, work-pulling and work-pushing
- architecture awareness, cost functions
- work in process:
  - goal: complete flexible skeleton library
  - alignment/semi-explicit placement
  - Placement Strategy brewer

Conclusion

Thank you!