Building Divide and Conquer From a Farm

Oleg Lobachev

Philipps Universität Marburg

Bad Honnef
May 2, 2011

Parallel Computer Algebra

- symbolic computation
- in a parallel functional language
- with algorithmic skeletons

. . . a Skeleton-Based Approach

- algorithmic skeletons = parallel algorithm abstractions
- in FP: higher-order functions
- skeletons as algorithm classification
- e. g., `map`-like, iteration, divide and conquer
- here: skeletons in same language as instantiation

⇒ focus on a special divide and conquer

Type of a Divide and Conquer Skeleton

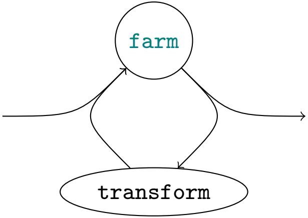
```
type DC a b = (a → Bool)
            → (a → b)
            → (a → [a])
            → ([b] → b)
            → a → b

farm :: (a → b) → [a] → [b]
```

Stream Processing Functions

- function on lists
- produces the result for initial list elements without waiting for further list elements
- works for infinite lists AKA *streams*

- `map (+1)` is stream processing
- `length` is not stream processing



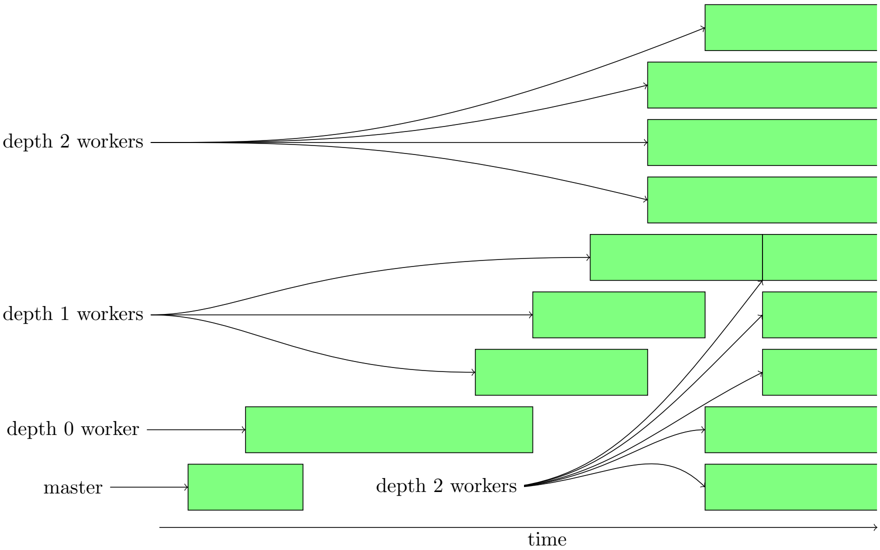
- `farm` = parallel `map` with task balancing
- process *divide* or *combine* tasks and send results back to `transform`
- at some point: *solve* locally in workers
- need an umbrella type

Umbrella type

- `RD a` = future for type `a`

```
data DCTask a b = InitialToDivide Depth a
                | ToDivide   Depth (RD a)
                | Divided    Depth [RD a]
                | Combined   Depth (RD b)
                | ToCombine  Depth [RD b]
```

Depth control I



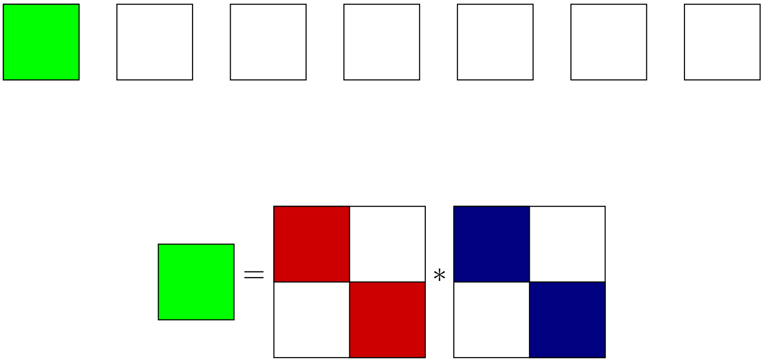
Depth control II

- depth for parallel divide
- depth for parallel combine
- depth for initial sequential divide
- maybe: depth for finalising sequential combine

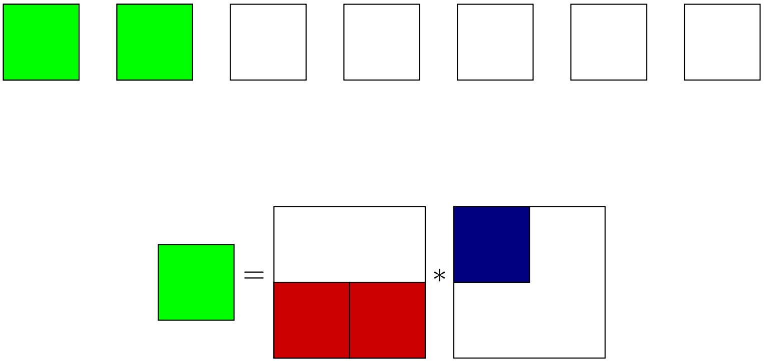
Other tuning parameters:

- arity of the DC tree

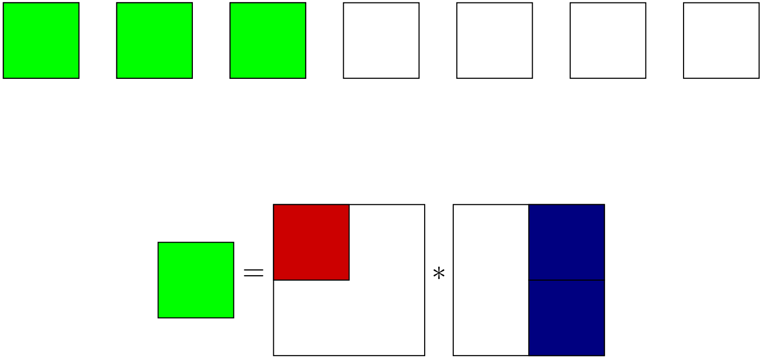
Strassen Multiplication: Divide



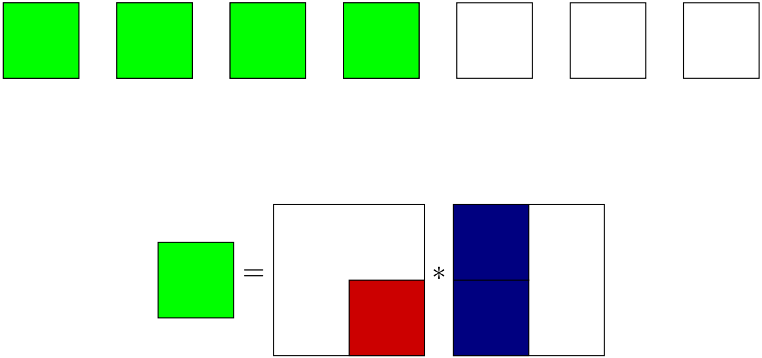
Strassen Multiplication: Divide



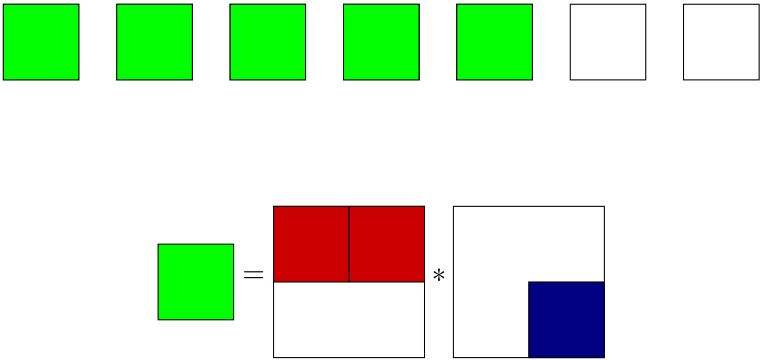
Strassen Multiplication: Divide



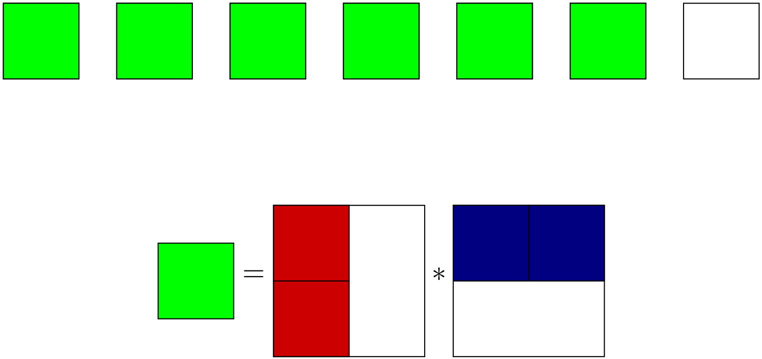
Strassen Multiplication: Divide



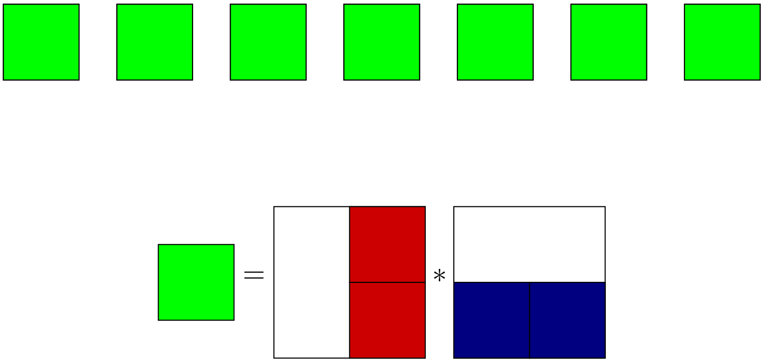
Strassen Multiplication: Divide



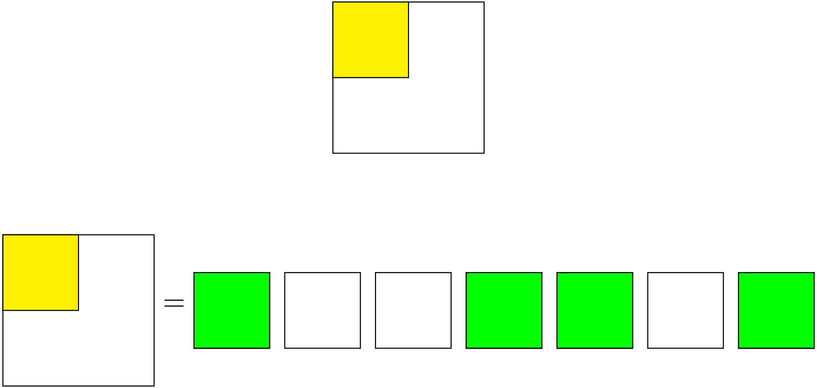
Strassen Multiplication: Divide



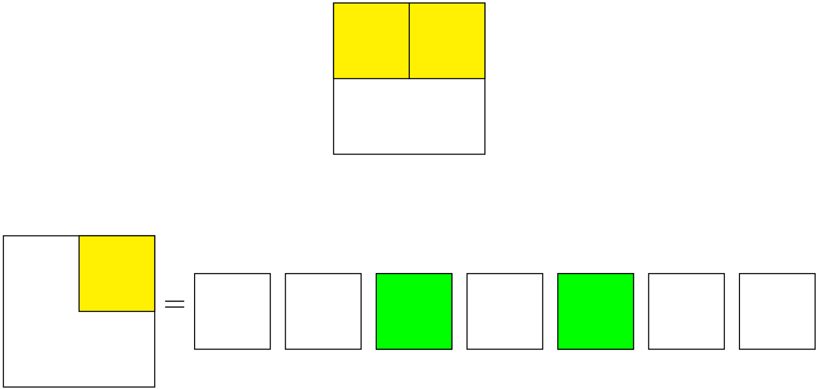
Strassen Multiplication: Divide



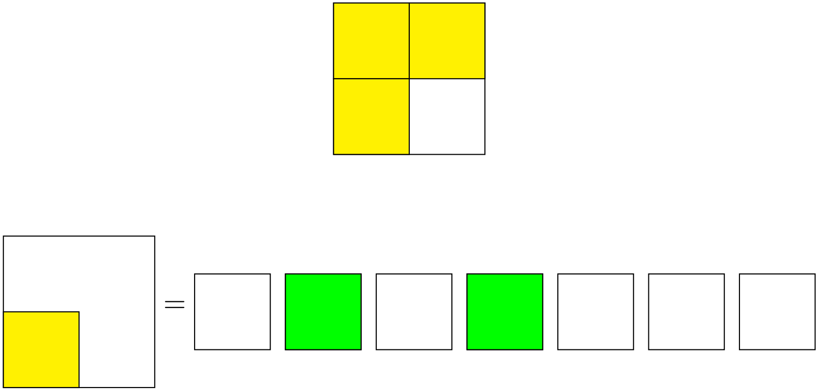
Strassen Multiplication: Combine



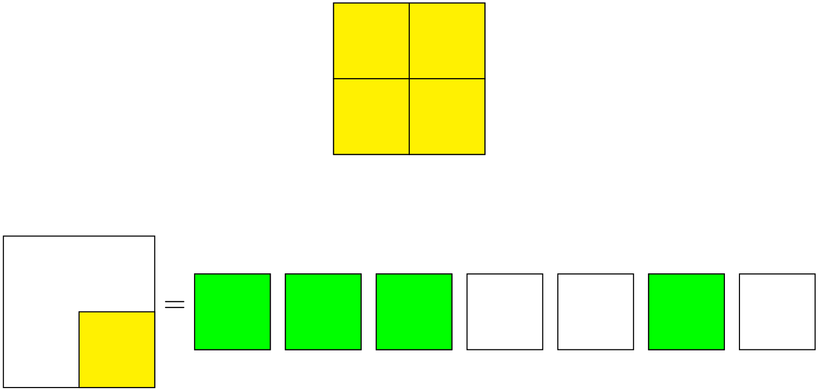
Strassen Multiplication: Combine



Strassen Multiplication: Combine



Strassen Multiplication: Combine



A Software Engineering Moment

- assume `divide`, `combine`, etc. as given
- sequential:

```
strassenSeq x y = dcSeq isTrivial solve divide combine (x, y)
```

A Software Engineering Moment

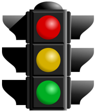
- assume `divide`, `combine`, etc. as given
- sequential:

```
strassenSeq x y = dcSeq isTrivial solve divide combine (x, y)
```

- parallel:

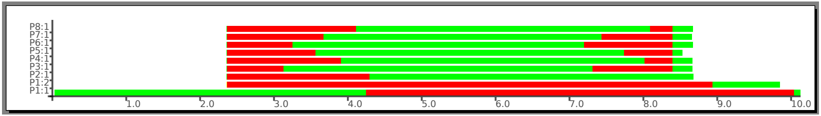
```
strassenPar x y = dcFarm 7 3 3 1 isTrivial solve divide combine (x, y)
```

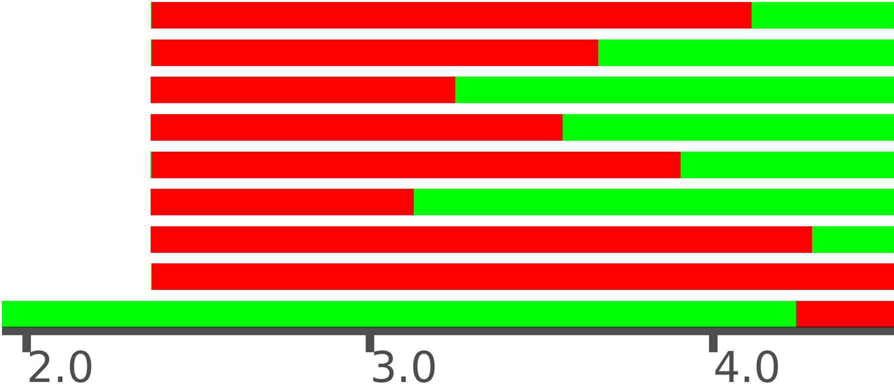
- *trace:* activity profile of a parallel program
- visualised as a diagaram
- horizontally: time, vertically: processor cores
- horizontal bars: processes



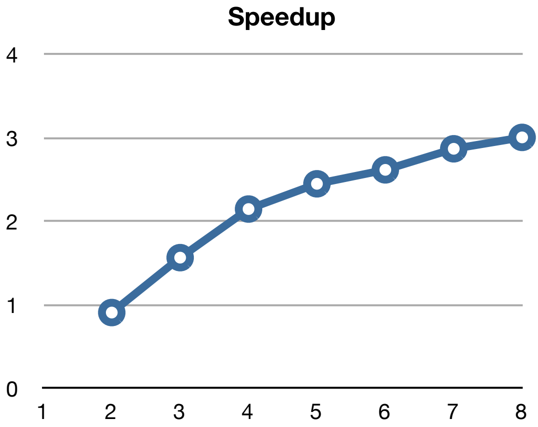
● red is blocked, ● yellow is runnable, ● green is running

Trace visualisation





Performance



- degrading speedup with larger depth
- worker disbalance
- sequential divide/combine is better?!
- is communication overhead to blame?
- try another use case?

Conclusions and Future Work

- skeletons = parallel h. o. f., drop-in replacements
- here: transformed DC to a *map*
- instantiated with Strassen multiplication

Conclusions and Future Work

- skeletons = parallel h. o. f., drop-in replacements
- here: transformed DC to a *map*
- instantiated with Strassen multiplication

- investigate concurrency problem with futures in initial steps
- worse performance at larger depth
- other, better instantiations?

Strassen Multiplication

- input A, B w. dimensions $2^l \times 2^l$, aim for: $C = AB$

$$\begin{aligned}
M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\
M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\
M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\
M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\
M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\
M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}),
\end{aligned} \tag{1}$$

- all multiplications in (1) with recursive calls

$$\begin{aligned}
C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\
C_{1,2} &= M_3 + M_5 \\
C_{2,1} &= M_2 + M_4 \\
C_{2,2} &= M_1 - M_2 + M_3 + M_6.
\end{aligned} \tag{2}$$

Code, dcFarmBody, transform I

```
type Depth = Int
type Arity = Int

data DCTask a b = InitialToDivide Depth a
                | ToDivide Depth (RD a)
                | Divided Depth [RD a]
                | Combined Depth (RD b)
                | ToCombine Depth [RD b]

-- NFData-Instanz
instance (NFData a, NFData b) ⇒ NFData (DCTask a b) where
    rnf (InitialToDivide  d v)  = rnf d ‘seq‘ rnf v
    rnf (ToDivide  d rd)  = rnf d ‘seq‘ rnf rd
    rnf (Divided   d rds) = rnf d ‘seq‘ rnf rds
    rnf (Combined  d rd)  = rnf d ‘seq‘ rnf rd
    rnf (ToCombine d rds) = rnf d ‘seq‘ rnf rds

-- Trans-Instanz
instance (Trans a, Trans b) ⇒ Trans (DCTask a b)
```

Code, dcFarmBody, transform II

```
catchNewToCombineTask :: Int → [DCTask a b]
                      → Maybe (DCTask a b, [DCTask a b])
catchNewToCombineTask k tasks =
     case splitAt k tasks of
          (tl@(t@(Combined l _):ts), resttl)
               →  if (all (isCombined l) ts)
                  then Just (ToCombine l
                             (map fromCombined tl), resttl)
                  else Nothing
          _ → Nothing
     where isCombined x (Combined y _) = x == y
           isCombined _ _  = False


-- the farm works lazily on the input list, thus creating a
-- angepasst f r Postfork-Parameter
dcFarmBody :: (Trans a, Trans b)
       ⇒ Arity
       → Depth -- ^ parallel depth
```

Code, dcFarmBody, transform III

```
       → Depth -- ^ postfork parameter
       → (Arity → [DCTask a b] → [DCTask a b])
          -- ^ task pool transform
       → (DCTask a b → DCTask a b) -- ^ working function
       → [DCTask a b] → [DCTask a b] -- ^ input to result
dcFarmBody k d postfork ttf f initTasks = localRes
    where -- paralleler Arbeitsanteil
          remoteRes = farm f (initTasks ++ newRemoteTasks)
          newRemoteTasks = ttf k putInPool

          -- Selektion ob Tasks parallel/sequentiell
          -- bearbeitet werden sollen
          (putInPool, stayLocal) =∘ ..

          -- lokaler, sequentieller Arbeitsanteil
          -- TODO: ohne RD machen
          localRes = stayLocal ++ map f newLocalTasks
          -- Verarbeitung von "stayLocal" schon erfolgt...
          newLocalTasks = ttf k localRes
```

Code, dcFarmBody, transform IV

```
partitionMy :: (a → Bool) → [a] → ([a], [a])
partitionMy p (x:xs) | p x = (x:ys, zs)
                     | otherwise = (ys, x:zs)
     where (ys,zs) = partitionMy p xs
partitionMy _ [] = ([],[])

{- transform - 'taskpool transform function' -}
transform :: Arity
          → [DCTask a b] → [DCTask a b]
            -- ^ task pool in and out
transform k ((Combined 0 x):r) = [] -- done!
transform k ((Divided d' xs):r) =
    let ys = zipWith ToDivide (repeat d') xs
    in ys ++ transform k r -- do the trick: flatten!
transform k [] = [] -- done! Postfork-Tiefe erreicht!
transform k xs = case catchNewToCombineTask k xs of
    Just (newToCombineTask, restTasks)
      → newToCombineTask : transform k restTasks
```

Code, DC Interface, WF I

```
dcFarm_ppfork :: forall a b. (Trans a, Trans b)
  ⇒ Arity -- ^ Arity des Divide-Baumes
  → Depth -- ^ Tiefe bis zu der parallel gearbeitet wird
  → Depth -- ^ ∘ .. allein der Master divide durchf hrt
  → Depth -- ^ ∘ .. nur noch im Master combined werden
  → (a → Bool) -- ^ is trivial?
  → (a → [a])  -- ^ divide
  → (a → b)    -- ^ solve
  → ([b] → b)  -- ^ combine
  → a → b      -- ^ input and result
dcFarm_ppfork k d pref postf isTr divide solve combine x
    = fetch $ fromCombined $ last $
        dcFarmBody k d postf transform (wf d) initT
      where initT = map (InitialToDivide splDepth) initRaw
            (initRaw, splDepth)
              = tryNtimes (concatMap divide)
                          (all (not ∘ isTr)) [x] pref
            -- Workerfunktion
            -- wf :: Depth → DCTask a b → DCTask a b
```

Code, DC Interface, WF II

```
-- Fall f r initialen Task (ohne RD)
wf d (InitialToDivide d' y)
    | isTr y = rnfAndModify ((Combined d')
               ∘ release) (solve y)
    | d' ≥ d = rnfAndModify ((Combined d')
              ∘ release) (dcSeq isTr divide
                solve combine y)
    | otherwise -- divide case
        = rnfAndModify (Divided (d'+1)
          ∘ releaseAll) (divide y)
-- normaler Fall
wf d (ToDivide d' x)
    | isTr y = rnfAndModify ((Combined d')
               ∘ release) (solve y)
    | d' ≥ d = rnfAndModify ((Combined d')
               ∘ release) (dcSeq isTr divide
                 solve combine y)
    | otherwise = rnfAndModify (Divided (d'+1)
                ∘ releaseAll) (divide y)
    where y = fetch x
```

Code, DC Interface, WF III

```
            -- Combine Fall
            wf d (ToCombine d' ys) = Combined (d'-1)
                                   $ (release ∘ combine
                                    ∘ fetchAll) ys

-- helper:
rnfAndModify :: NFData a ⇒ (a → b) → a → b
rnfAndModify f x = rnf x `seq` f x

-- RD interface:
release :: a → RD a
fetch :: RD a → a
releaseAll :: [a] → [RD a]
fetchAll :: [RD a] → [a]
```