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
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

\[
\text{type } \text{DC } a \ b = (a \rightarrow \text{Bool}) \\
\quad \rightarrow (a \rightarrow b) \\
\quad \rightarrow (a \rightarrow [a]) \\
\quad \rightarrow ([b] \rightarrow b) \\
\quad \rightarrow a \rightarrow b
\]

\[
\text{farm :: (a } \rightarrow b) \rightarrow [a] \rightarrow [b]
\]
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 \( a \)

\[
\text{data} \ DCTask \ a \ b = \ \text{InitialToDivide} \ \text{Depth} \ a \\
| \ \text{ToDivide} \ \text{Depth} \ (RD \ a) \\
| \ \text{Divided} \ \text{Depth} \ [RD \ a] \\
| \ \text{Combined} \ \text{Depth} \ (RD \ b) \\
| \ \text{ToCombine} \ \text{Depth} \ [RD \ b]
\]
Depth control I

- Depth control
- Types
- Streams
- Idea
- DCTask
- Master
- Depth 0
- Worker
- Depth 1
- Worker
- Depth 2
- Worker
- Depth 2
- Workers
- Time

Oleg Lobachev
Building Divide and Conquer From a Farm
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

\[
\begin{array}{ccc}
\text{Green} & \text{Red} & \text{Red} \\
\text{White} & \text{White} & \text{Red} \\
\text{Red} & \text{White} & \text{Blue} \\
\end{array}
\]

\[= \]

\[
\begin{array}{ccc}
\text{Green} & \text{White} & \text{Red} \\
\text{Red} & \text{White} & \text{Blue} \\
\end{array}
\]

\[\ast\]
Strassen Multiplication: Divide
Strassen Multiplication: Divide
Strassen Multiplication: Divide

\[
\begin{array}{c}
\begin{array}{c}
\text{Green square} \\
\text{Red rectangle} \\
\text{Blue rectangle}
\end{array}
\end{array}
\]

\[=\]

\[
\begin{array}{c}
\text{Green square} \\
\text{Cyan rectangle}
\end{array}
\]

\[\times\]
Strassen Multiplication: Divide

\[ \begin{array}{cccc}
\text{Image 1} & \text{Image 2} & \text{Image 3} & \text{Image 4} \\
\text{Image 5} & \text{Image 6} & \text{Image 7} & \text{Image 8}
\end{array} \]

\[ \begin{array}{ccc}
\text{Image 9} & \text{Image 10} & \text{Image 11} \\
\text{Image 12} & \text{Image 13} & \text{Image 14}
\end{array} \]

\[ \begin{array}{c}
\text{Image 15} \\
\text{Image 16}
\end{array} \]
Strassen Multiplication: Divide

\[
\begin{array}{c}
\quad \quad \quad \quad \\
\end{array}
\]
Strassen Multiplication: Divide
Strassen Multiplication: Combine

\[
\begin{bmatrix}
0 & 1 \\
2 & 0
\end{bmatrix}
\begin{bmatrix}
3 & 1 \\
0 & 4
\end{bmatrix} =
\begin{bmatrix}
12 & 7 \\
10 & 2
\end{bmatrix}
\]
Strassen Multiplication: Combine

=
Strassen Multiplication: Combine

\[
\begin{array}{cc}
\text{Divide} & \text{Combine} \\
\text{SE Part} & \text{Trace} & \text{Performance}
\end{array}
\]
Strassen Multiplication: Combine

=
A Software Engineering Moment

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

\[
\text{strassenSeq } x \ y = \text{dcSeq isTrivial solve divide combine } (x, y)
\]
A Software Engineering Moment

- assume divide, combine, etc. as given
- sequential:
  \[
  \text{strassenSeq } x \ y = \text{dcSeq isTrivial solve divide combine } (x, y)
  \]
- parallel:
  \[
  \text{strassenPar } x \ y = \text{dcFarm 7 3 3 1 isTrivial solve divide combine } (x, y)
  \]
• *trace*: activity profile of a parallel program
• visualised as a diagram
• 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?

Speedup

1 2 3 4 5 6 7 8

dcfarm ppfork 7.0 pe t seq speedup
1 27,015,870
2 29,607,253 27,015,870 0.9124747236767
3 17,204,078 27,015,870 1.5703178048832
4 12,551,124 27,015,870 2.1524661854986
5 11,004,699 27,015,870 2.4549394763092
6 10,302,881 27,015,870 2.6221665571018
7 9,398,690 27,015,870 2.8744293087654
8 8,972,319 27,015,870 3.0110242402215

Oleg Lobachev
Building Divide and Conquer From a Farm
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: $C = AB$

\[
\begin{align*}
M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\
M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\
M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\
M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\
M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\
M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}),
\end{align*}
\]

- all multiplications in (1) with recursive calls

\[
\begin{align*}
C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\
C_{1,2} &= M_3 + M_5 \\
C_{2,1} &= M_2 + M_4 \\
C_{2,2} &= M_1 - M_2 + M_3 + M_6.
\end{align*}
\]
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-Instanzi
instance (Trans a, Trans b) ⇒ Trans (DCTask a b)

Oleg Lobachev  Building Divide and Conquer From a Farm
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
→ 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

Oleg Lobachev

Building Divide and Conquer From a Farm
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
dcFarm_ppfork :: forall a b. (Trans a, Trans b)
⇒ Arity -- ^ Arity des Divide-Baumes
⇒ Depth -- ^ Tiefe bis zu der parallel gearbeitet wird
⇒ (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

Oleg Lobachev
Building Divide and Conquer From a Farm
-- Fall für initialen Task (ohne RD)
\[\text{wf } d \text{ (InitialToDivide } d' \text{ y)}
\mid \text{isTr } y = \text{rnfAndModify } ((\text{Combined } d')
\mid \text{release}) (\text{solve } y)
\mid d' \geq d = \text{rnfAndModify } ((\text{Combined } d')
\mid \text{release}) (\text{dcSeq isTr divide solve combine } y)
\mid \text{otherwise} -- divide case
\mid = \text{rnfAndModify } (\text{Divided } (d' + 1)
\mid \text{releaseAll}) (\text{divide } y)
\]

-- normaler Fall
\[\text{wf } d \text{ (ToDivide } d' \text{ x)}
\mid \text{isTr } y = \text{rnfAndModify } ((\text{Combined } d')
\mid \text{release}) (\text{solve } y)
\mid d' \geq d = \text{rnfAndModify } ((\text{Combined } d')
\mid \text{release}) (\text{dcSeq isTr divide solve combine } y)
\mid \text{otherwise} = \text{rnfAndModify } (\text{Divided } (d' + 1)
\mid \text{releaseAll}) (\text{divide } y)
\]

where \( y = \text{fetch } x \)
-- Combine Fall
wf d (ToCombine d' ys) = Combined (d'-1)
 $ (\text{release} \circ \text{combine} \circ \text{fetchAll}) \ ys$

-- helper:
\( \text{rnfAndModify} \) :: NFData a \Rightarrow (a \rightarrow b) \rightarrow a \rightarrow b
\text{rnfAndModify} \ f \ x = \text{rnf} \ x \ \text{`seq`} \ f \ x

-- RD interface:
\text{release} :: a \rightarrow \text{RD} \ a
\text{fetch} :: \text{RD} \ a \rightarrow a
\text{releaseAll} :: [a] \rightarrow [\text{RD} \ a]
\text{fetchAll} :: [\text{RD} \ a] \rightarrow [a]