Faculty of Science

Segmented Scans and Nested Data Parallelism

Andrzej Filinski
andrzej@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

Dyalog APL Conference

15 October 2012, Helsingør, Denmark

A bit of context

- Andrzej Filinski, Associate Professor at UCPH.
- Member of the APL Research Group ...
  - **A**lgorithms and **P**rogramming **L**anguages
  - But acronym clash not entirely coincidental.
- ... & HIPERFIT Research Center
  - "Functional High-Performance Computing for Financial Information Technology"
  - Key interest: functional, array-oriented languages as high-level programming paradigm for massively parallel computing platforms (many-core, GPGPUs, FPGAs, ...)
  - Working with Dyalog to do bring some of this technology to the real world.
- **Important disclaimer:** I am not a real APL programmer!
  - Dabbled a bit in APL/370 some 30 years ago.
  - Ignorant of many common idioms and Dyalog APL features
  - Hopefully the underlying ideas will still come through, even if less elegantly than what you are used to.

Parallelism and concurrency

- Parallelism $\neq$ concurrency
  - Concurrency: explicitly dealing with things happening at once (threads, synchronization, communication, etc.)
    - Still relevant on single CPU with time slicing
  - **Parallelism**: obtaining result faster (in wall-clock time) by exploiting multiple computation units.
    - No need for exposing concurrency to programmer.
- APL is not a parallel *language*.
  - No parallel cost model/semantics
  - But eminently suitable for parallel *implementation*.
    - Especially data (as opposed to control) parallelism.
- This talk: efficient *nested* data parallelism for array-oriented languages
  - Based on Guy Blelloch's work in early 1990s.
    - Targeted the Connection Machine: decades ahead of its time.
  - Same ideas now also being explored in, e.g., DP Haskell.

APL and data parallelism

- APL has seemingly very parallel(izable) execution model.
  - Element-wise primitive operations: `+`, `<`, `*`, ...
  - Gather/scatter primitives (`←v[is]`, `v[is]←`).
  - Uniform/regular bulk operations: `⍳`, `⍉`, `,`, ...
  - "Embarrassingly parallel"
- But some operations seem inherently sequential:
  - Cumulative *data* dependencies: scan (`+\`), ...
  - Cumulative *index* dependencies: compress (`/`), ...
- Even nominally independent computations (*F*¨) have their own challenges wrt. parallelism:
  - Control flow (`→`, `:`) in *F* precludes SIMD-style parallelization
  - Poor load balancing: `{+/⍳⍵}¨2 6 50 3 4 6`
- There *is* a magic bullet:
  - "Swiss army chainsaw" of parallel algorithms: segmented (aka. partitioned) scans.

Parallel prefix sums (+-scans)

- Paradigmatic parallel-computing problem: Given a long (say, 10^6 elts) numeric vector `V`, compute `+\V`,
  - **Note:** how would want the APL system to implement `+\V`, not how you'd want to re-express the scan in APL yourself.
- Suppose one addition takes 1 ns; ignore memory access and control overhead for now.
- Sequential algorithm (`for`/`do`-loop with accumulator in C/Fortran, `foldl` in Haskell/ML): $10^6 \times 1$ ns $= 1$ ms.
- Now suppose we have 1000 cores (e.g., large GPU). How fast can we do it?
  - Optimistic answer (lower bound): 1000 times faster, i.e., 1 μs.
  - Pessimistic answer (upper bound): data dependency creates sequential bottleneck, so no speedup; still 1 ms.
  - The true answer lies somewhere in between...

A simple parallel scan algorithm

- Exploits essentially that addition is *associative*.
  - But not that commutative or invertible.
- Three phases:
  1. Partition vector into 1000 blocks of 1000 elements each. Independently scan each block (1000 ns = 1 μs, using all 1000 processors):

3	1	4	1	5	9	2	6	5
3	4	8	1	6	15	2	8	13

  2. Collect last elements of block scans, and scan them (1 μs, using one processor):

8	15	13
8	23	36

  3. Use each result to adjust next block's scans (1 μs, using 999 processors):

3	4	8	1	6	15	2	8	13
			8	8	8	23	23	23
3	4	8	9	14	23	25	31	36

- Total: 2×10^6 additions, ~ 3 μs. Not too bad, but middle phase is still disturbingly sequential...

Unite-and-conquer scan

- Practical and adaptive parallel algorithm
  - Also useful in sequential settings: exploits vectorized primitives
- Example, for power-of-two vector length:

```
v                        3  1  4  1  5  9  2  6
o ← 1⌷⍉((.5×⍴v),2)⍴v     3     4     5     2
e ← 2⌷⍉((.5×⍴v),2)⍴v        1     1     9     6
p ← o+e                     4     5    14     8
s ← ∇ p                     4     9    23    31
r ← (1⌷o),(¯1↓s)+1↓o     3     8    14    25
w ← ,r,[1.5]s            3  4  8  9 14 23 25 31
```

- Total of $\log_2 n$ recursive calls for length-n vector.
- Total of $2\frac{n}{2} + 2\frac{n}{4} + \cdots + 2 \simeq 2n$ element additions.
- For arbitrary operations and vector sizes (but still rank-1 only):

```
PSCAN ← {(⍴,⍵)≤1:⍵ ⋄ (o e) ← ↓⍉((⌈.5×⍴⍵),2)⍴⍵ ⋄
            s ← ∇ o⍺⍺¨e ⋄ r ← (1⌷o),(¯1↓s)⍺⍺¨1↓o ⋄
            (-2|⍴⍵)↓,r,[1.5]s}
```

Aside: Sequential performance of scans

- Common case: base operation also works vectorized (like +).
  - Optimized `VPSCAN`: like `PSCAN`, but with `αα` in place of `αα¨`.
- APL's native scan is right-too-left.
  - Quadratic running time: prohibitively expensive for more than a few thousand elements.
  - Special case for + and other associative primitives, but doesn't cover `{α+ω}`, or more exotic, programmer-defined functions.
- A few quick performance tests on a small machine:
  - `{α+ω} \ ⍳1E6`: near-infeasible (a few days, extrapolated).
  - `{α+ω} PSCAN ⍳1E6`: takes about 1 second.
  - `{α+ω} VPSCAN ⍳1E6`: takes about 60 ms.
  - `+\ ⍳1E6`: takes about 25 ms.
- Reflects that parallel algorithm does twice as much work, but most of it in huge chunks.

Sequential performance, continued

- From `http://dfns.dyalog.com/c_ascan.htm`:

```
ascan ← {⎕ML←0 ◊ 2>0⊥⍴⍵:⍵ ◊
            ⌽↑⍺⍺{(⊂(⊃⍵)⍺⍺ ⍺),⍵}/⌽(⊂∘⊃¨↓⍵),↑1↓¨↓⍵}
```

  - Repeatedly extends vector by one element: ultimately also quadratic behavior, but hits the performance wall a bit later.
  - `{⍺+⍵} ascan ⍳1E6`: about 15 minutes (extrapolated).
- Unlike the others, `VPSCAN` is also trivially parallelizable.
  - Only needs efficient vector addition (+ some data movement).
- In practice, parallel speedups are less than what algorithmic complexity would suggest, but still worthwhile.
  - Efficient, hand-tuned implementation of scans exist for CUDA (NVIDIA GPUs), multiple HPC libraries.
  - Use basically the unite-and-conquer algorithm above, though hard to see from the C code.

Why care so much about fast scans?

- Key to parallel implementation of lots of other primitives
- (Inside processor: look-ahead-carry adders do scans in hardware.)
  - Essential for, e.g., 64-bit arithmetic.
  - Or for parallelizable bignum packages (RSA crypto, etc.)
- Reduction: unite-and-conquer algorithm can be simplified a bit if we only want the final result:
  - Assumes non-empty vector:

    ```
    PREDUCE ← {(⍴,⍵)≤1:⊃⍵ ⋄ (o e)←↓⍉((⌊.5×⍴⍵),2)⍴⍵ ⋄
                   ∇ (o ⍺⍺¨ e),(-2|⍴⍵)↑⍵}
    ```
  - VPREDUCE variant with just ⍺⍺ instead of ⍺⍺¨.
  - Performs only as many basic operations as vector length.
  - {⍺+⍵} VPREDUCE a bit faster than {⍺+⍵}/, but much slower than simple +/.
- But efficient scans are also the key to parallelizing lots of other, seemingly sequential, tasks.

Uses of scans II: compress, flag-partition

- Given data vector `v`, flag vector `f`, with `ρv=ρf`; compute `w ← (f/v),(~f)/v`.
- Example:

(index)	1 2 3 4 5 6 7 8
`v`	*3 1 4 1 5 9 2 6*
`f`	1 1 0 0 1 0 1 1
`s ← +\f`	1 2 2 2 3 3 4 5
`ns ← (⍳ρf)+s[ρs]-s`	5 5 6 7 7 8 8 8
`a ← (s×f)+ns×~f`	1 2 6 7 3 8 4 5
`w ← ?(ρf)ρ42`	*? ? ? ? ? ? ? ?*
`w[a] ← v`	*3 1 5 2 6 4 1 9*

- Only one scan; all other operations are trivially parallelizable
- If we only need `f/v` or `(~f)/v`, just take appropriate slice of `w`.
- Replicate (`/` with non-boolean flags): see later.

Uses of scans III: expand, flag-merge

- Flag vector `f`, data vectors `v1` and `v2`, with `(ρv1)+ρv2` = `ρf`; compute `w ← (f\v1) + (~f)\v2`
(index)	1 2 3 4 5 6 7 8
`v1`	*3 1 5 2 6*
`v2`	*4 1 9*
`f`	1 1 0 0 1 0 1 1
`v ← v1,v2`	*3 1 5 2 6 4 1 9*
`a ←` (from `f` as before)	1 2 6 7 3 8 4 5
`w ← v[a]`	*3 1 4 1 5 9 2 6*
- **Note:** no actual addition; works for non-numeric data as well.
  - `f\v` or `(~f)\v` by itself easily expressible as flag-merge with zero or blank vector, as appropriate.
- Permutation `a` depends only on `f`: Single +-scan of `f` enables all four functions: `f/`, `(~f)/`, `f\`, and `(~f)\`.

What about parallelizing control flow, or recursion?

- First step: the *vectorization* transformation.
- `FACT ← {ω=0:1 ◊ ω×∇ ω−1}` (like `!`, but ω must be scalar)
  - General pattern: `F ← {P ω:B ω ◊ ω C (∇ R ω)}`, where `P={ω=0}`, `B={1}`, `C={α×ω}`, `R={ω−1}`
- Goal: define `FACTV` s.t. `FACTV v` ↔ `FACT¨v`.
- ```
  FACTV ← {0=ρ,ω:⍬ ◊ f←ω=0 ◊ r←(~f)/ω ◊
                (f\1) + (~f)\r×∇(r−1)}
  ```
- **Note**: *total* of `⌈/ω` recursive calls.
- Performance test: `FACTV` about 30 times faster than `FACT¨` on `?1E5ρ100`.
  - Again, with parallel back end, should do even better.
- Same transform works for all functions using that general pattern.
  ```

Eliminating redundant work

- Can easily filter duplicate requests to vectorized functions.
  - `UMAP ← {u←∪⍵ ⋄ (⍺⍺ u)[u⍳⍵]}`
  - Invariant: `FV UMAP v` ↔ `FV v`, but faster.
- Not unlike memoization, dynamic programming, but in space rather than time:
  - Memoization: have I been asked this before?
  - Duplicate trimming: am I being asked the same thing twice?
- Performance note: algorithmically, this `UMAP` is a bit dubious.
  - ∪ is presumably implemented well, but the ⍳ could take quadratic time, unless the interpreter is very clever.
  - Proper solution would probably involve explicit sorting, or hashing of ⍵.
- Can add `UMAP` outside, or inside, `FACTV`.

Simple nested parallelism

- `FIB ← {ω≤1:ω ◊ (∇ ω-1)+(∇ ω-2)}`
  - Pattern: {*P* ω:*B* ω ◊ ω *C* (∇ R_1 ω) (∇ R_2 ω)}
- Explicating potential for data parallelism:
  `FIBP ← {ω≤1:ω ◊ +/∇¨(ω-1) (ω-2)}`
- `FIBV ← {0=ρ,ω:⍬ ◊ f←ω≤1 ◊ r←(~f)/ω ◊`
  `(f\f/ω) + (~f)\+⌿(2,ρr)ρ(∇(r-1),(r-2))}`
  - Trading space for time: in recursive call, argument vector is twice as long as input vector.
- Vectorization exposes massive potential for speedup.
  - Even if original argument vector is duplicate-free, vectorized recursive calls create lot of redundancies:
- `FIBVU ← {0=ρ,ω:⍬ ◊ f←ω≤1 ◊ r←(~f)/ω ◊`
  `(f\f/ω) + (~f)\+⌿(2,ρr)ρ(∇ UMAP (r-1),(r-2))}`
  - Can now easily compute `FIBVU ?1000ρ1000`.
  - Space usage “only” quadratic, not exponential.

Segmented scans

- A harder challenge: Still 10^6 elts total, but partitioned into nested vectors; compute scan independently for each segment:
  `+\¨(3 1 4) (1 5 9 2) (6) (5 4)` $\leftrightarrow$
  `(3 4 8) (1 6 15 17) (6) (5 9)`
- Some segments may be very long (e.g., 10^5 elements); a lot may be very short (e.g., 10^4 length-10 segments), in an unpredictable pattern.
- Should work for any associative operation (e.g., ⌈), not necessarily invertible: can't just compute unsegmented scan, then adjust by subtraction.
- Straightforward sequential implementation: time proportional to total length + number of segments.
- How to implement efficiently in parallel on 1000 processors?

Implementing segmented scans

- Represent vector explicitly as data + leading partition flags
  v ← 3 1 4 1 5 9 2 6 5 4
  p ← 1 0 0 1 0 0 0 1 1 0
  p ⊂ v ↔ (3 1 4) (1 5 9 2) (,6) (5 4)
- Consider operation: $\langle {a \atop p} \rangle \oplus \langle {b \atop q} \rangle = \langle {a\times\tilde{q}+b \atop p\vee q} \rangle$ ($\tilde{p}$ is negation).
- Top row precisely expresses desired behavior of segmented left-to-right +-scan:
  - Either add to accumulator, or reset it, depending on flag.
- $\oplus$ is associative: $(\langle {a \atop p} \rangle \oplus \langle {b \atop q} \rangle) \oplus \langle {c \atop r} \rangle = \langle {(a\times\tilde{q}+b)\times\tilde{r}+c \atop (p\vee q)\vee r} \rangle =$ $\langle {a\times\tilde{q}\times\tilde{r}+b\times\tilde{r}+c \atop p\vee q\vee r} \rangle = \langle {a\times\tilde{(q\vee r)}+(b\times\tilde{r}+c) \atop p\vee(q\vee r)} \rangle = \langle {a \atop p} \rangle \oplus (\langle {b \atop q} \rangle \oplus \langle {c \atop r} \rangle)$
  - So can use the parallel algorithm to compute $\oplus$-scan!
- `FPLUS ← {(a p)←α ◊ (b q)←ω ◊ ((a×~q)+b) (p∨q)}`
- `FPLUSV ← {(a p)←↓⍉↑α ◊ (b q)←↓⍉↑ω ◊ ↓⍉↑((a×~q)+b) (p∨q)}`

Implementing segmented scans II

- `SPLUSSCAN ← {⊃¨FPLUSV VPSCAN ↓⍉↑ω}`
  - For illustration purposes only; want to keep data and flags as separate vectors, rather than vector of pairs.
- Invariant: `+\¨p⊂v` ↔ `p⊂SPLUSSCAN (v p)`.
- For *any* associative •, define $\left\langle {a \atop p} \right\rangle \odot \left\langle {b \atop q} \right\rangle = \left\langle {((a \bullet b), b)[1+q] \atop p \vee q} \right\rangle$
  - Then $\odot$ also associative, though a bit harder to see.
- Systematically obtain segmented versions of derived primitives (reduce, compress, ...)
  - Note: segmented •-reduce needs $\odot$-scan, not just $\odot$-reduce.
- Also: segmented ⍳, ⍴, etc.
  - Example: replicate (/) can be expressed as segmented ⊣-scan.
- Can now efficiently parallelize, e.g, `{+/(⍳ω)*2}¨2*?100⍴20`
- (Final ingredient: *streaming*; avoid materializing entire nested vector at once, but compute in chunks.)

General nested data parallelism

- An actually useful recursive algorithm:

```
QSORT ← {(ρω)≤1:ω ◊ p←ω[⌈.5×ρω] ◊
          (∇ (ω<p)/ω),((ω=p)/ω),(∇ (ω>p)/ω)}
```

- Same recursion pattern as `FIB`, but with whole vectors as data values; compress, concatenate, etc. instead of arithmetic.
- Because all these primitives definable in terms of scan, they work directly with segment flags, too.
- Hand-vectorized version (`QSORTV`) quite messy, but whole point is that the transformation can be automated.
- (Expected) log *n* recursive calls *total*.
  - Global control flow still handled by interpreter
  - All the actual work (<, /, ,) still done in bulk by vectorized primitives.
    - Possibly off-loaded to compute accelerator (GPU, etc).

Parallel algorithms

- APL like Perl: "There's more than one way to do it..."
  - "... but most of them suck."
  - There's only so much a clever compiler can do with a quadratic (or worse) computation specification.
  - Even more insidious: algorithm (or idiom!) may behave fine sequentially, but be fundamentally unparallelizable.
- "Functional [and APL] programmers know the value of everything, bot the cost of nothing."
  - Need at least some cost awareness: understand both *work* and *depth* complexity of chosen (sub)algorithm.
- Algorithms matter, even (especially?) in an array language.
  - Exploit algebraic properties that are not apparent to compiler (associativity of operations, sortedness of vectors, etc.)
- (Segmented) scans are not the only trick in the parallel algorithms book!
  - Mainly used to provide data-parallel substrate, to allow expression of data-parallel programs like `QSORT`.

Summary and final remarks

- Parallel platforms are coming whether we want them or not.
  - Processor speeds essentially stagnant, but core counts steadily increasing.
  - Element-wise processing becoming fundamentally untenable.
- Goal of the language should be to support programmer in expressing parallel computations *naturally*.
  - APL is an excellent match, but with a few pitfalls.
  - Compiler can do a lot, but program must be parallelism-aware.
- Scans are cool. Really.
- Basic data parallelism (vectorized primitives) good, nested data parallelism better.
  - Fine-grained "each" ($F^{\text{¨}}$) has lots of potential, but requires considerable subtlety to implement effectively.
  - We're working on it...
- It's an exciting time to be an array programmer!