Data Parallel Haskell

SP Jones, R Leshchinskiy,
G Keller, M Chakravarty

Slides partially pilfered from Simon Peyton Jones

Motivation

Multicore

Parallel programming essential

Task parallelism

- Explicit threads
- Synchronise via locks, messages, or STM

Modest parallelism
Hard to program

Data parallelism

Operate simultaneously on bulk data

Massive parallelism
Easy to program

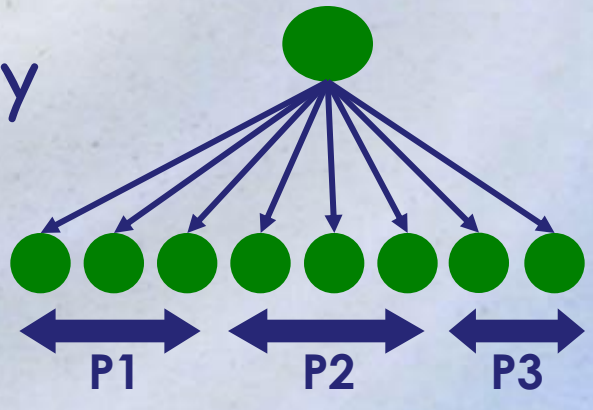
- Single flow of control
- Implicit synchronisation

Flat data parallelism

- Apply sequential operation to bulk data
- Widely used, well understood, well supported

```
foreach i in 1..N {
    ...do something to A[i]...
}
```

- "something" is sequential
- Single point of concurrency
- Great for dense matrices



1,000,000's of (small) work items

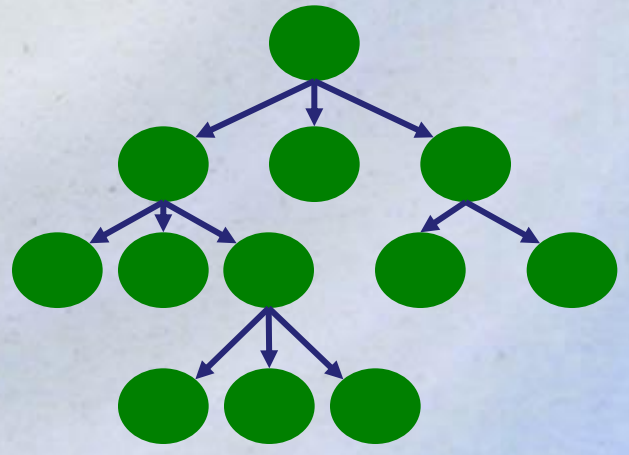
e.g. HPF, MPI, MapReduce, Matlab's Parallel Toolbox

Nested data parallel

- Main idea: **allow "something" to be parallel**

```
foreach i in 1..N {
    ...do something to A[i]...
}
```

- Now the parallelism structure is recursive, and un-balanced
- Hard to implement!



Still 1,000,000's of (small) work items

Nested DP is great for **programmers**

- Fundamentally more modular
- Opens up a much wider range of applications:
  - Sparse arrays, variable grid adaptive methods (e.g. Barnes-Hut)
  - Divide and conquer algorithms (e.g. sort)
  - Graph algorithms (e.g. shortest path, spanning trees)
  - Physics engines for games, computational graphics (e.g. Delauny triangulation)
  - Machine learning, optimisation, constraint solving

Nested DP is tough for **compilers**

- ...because the concurrency tree is both irregular and fine-grained
- But it can be done! NESL (Blelloch 1995) is an existence proof
- Key idea: "flattening" transformation:
  nested DP ➡ flat DP

Data parallelism **requires** functional programming

```
B = true;
Sum = 0;
A = [1,2,3,4,5];
foreach i in 1..length(A) {
    B = not(B);
    if (B) then Sum = Sum + A[i]
}
```

- **Side effects must be synchronized!**
- Much harder to implement
- Synchronization kills performance

Why Data Parallel **Haskell**?

- Purely functional language
  - *Computations don't affect shared state unless indicated by their type*
  - *Execution order only constrained by data dependencies when specified by programmer*
- Rich type system
  - *helps hide implementation details*
- Mature and speedy compiler (GHC)
  - *easy to specify source-to-source transformations*

What does DPH add?

- Parallel arrays
  - *efficient unboxed representations for all Haskell data types*
- Parallelized library of array operations
- Convenient array comprehension syntax
- Source-to-source transformation pipeline
  - *Turns inefficient Nested Data parallelism into optimized Flat Data parallelism*
- "Gang Parallelism" execution model

Parallel Arrays

```
class ArrElem e where
  data [:e:]
       Parallel array is associated with ArrElem
       type class. This allows us to specify
       efficient array implementations for different
       element types.
  (!:) :: [:e:] -> Int -> e
       Array indexing.
  mapP :: (a->b) -> [:a:] -> [:b:]
       Maps first argument over elements of parallel
       array
  replicateP :: Int -> a -> [:a:]
       Generate parallel array by replicating second
       argument
  etc...
```

Parallel Arrays - Example

```
class ArrElem Int where
  data [:Int:] = ArrInt ByteArray
       Efficiently stores [:Int:] as a dense unboxed
       byte array.
  (!:) (ArrInt a) i = indexIntArray a i

  mapP fn a = mapIntArray fn a

  replicateP n x = replicateIntArray n x

  etc...
```

Note: These **aren't** parallel operations, but instead specify what happens at each core.

Parallel array comprehensions

[:Float:] is the type of parallel arrays of Float

```
vecMul :: [:Float:] -> [:Float:] -> Float
vecMul v1 v2 = sumP [: f1*f2 | f1 <- v1 | f2 <- v2 :]
```

sumP :: [:Float:] -> Float

An array comprehension: "the array of all f1*f2 where f1 is drawn from v1 and f2 from v2"

Operations over parallel array are computed in parallel; that is the only way the programmer says "do parallel stuff"

NB: no locks!

Sparse vector multiplication

A sparse vector is represented as a vector of (index,value) pairs

```
svMul :: [:(Int,Float):] -> [:Float:] -> Float
svMul sv v = sumP [: f*(v !: i) | (i,f) <- sv :]
```

v !: i gets the i'th element of v

Parallelism is proportional to length of sparse vector

Sparse matrix multiplication

A sparse matrix is a vector of sparse vectors

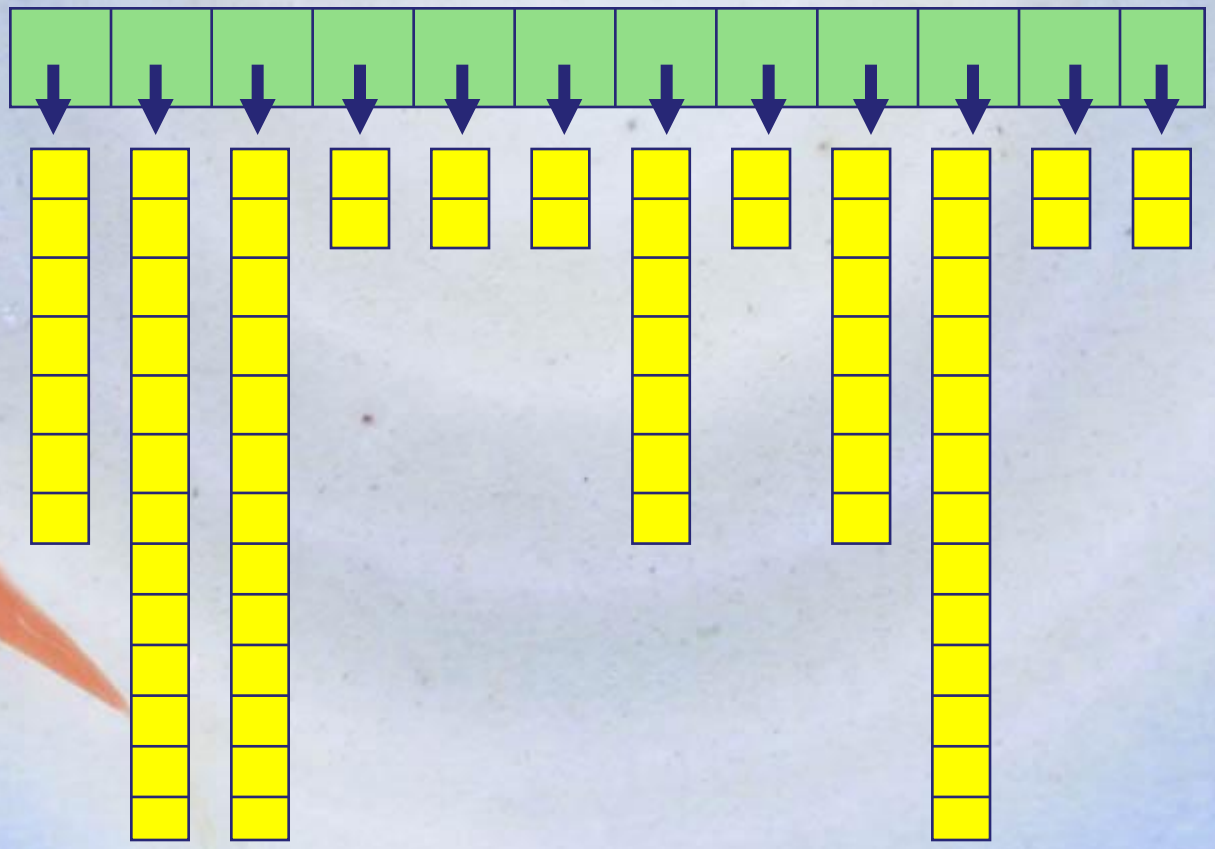
```
smMul :: [:[:(Int,Float):]:] -> [:Float:] -> Float
smMul sm v = sumP [: svMul row v | row <- sm :]
```

Nested data parallelism here!

We are calling a parallel operation, svMul, on every element of a parallel array, sm

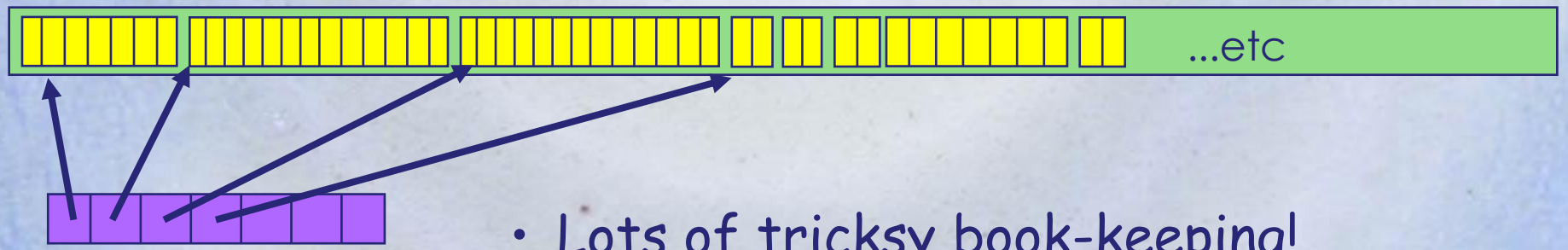
Hard to implement well

- Evenly chunking at top level might be **ill-balanced**
- Top level along might **not be very parallel**



The flattening transformation

- Concatenate sub-arrays into one big, flat array
- Operate in parallel on the big array
- Segment vector keeps track of where the sub-arrays are



- Lots of tricksy book-keeping!
- Possible to do by hand (and done in practice), but very hard to get right
- Blelloch's NESL showed it could be done systematically

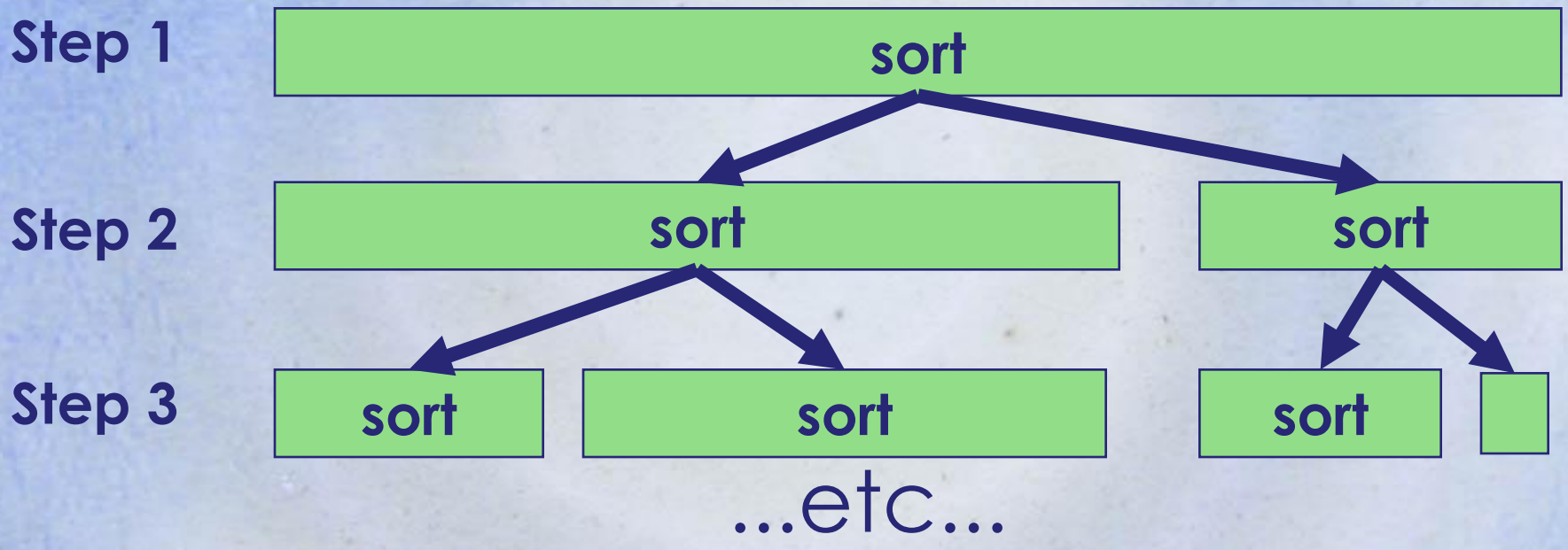
Data-parallel quicksort

```
sort :: [:Float:] -> [:Float:]
sort a = if (length a <= 1) then a
         else sa!:0 +++ eq +++ sa!:1
   where
      m = a!:0
      lt = [: f | f<-a, f<m :]
      eq = [: f | f<-a, f==m :]
      gr = [: f | f<-a, f>m :]
      sa = [: sort a | a <- [:lt,gr:] :]
```

Parallel filters

2-way nested data parallelism here!

How it works



- All sub-sorts at the same level are done in parallel
- Segment vectors track which chunk belongs to which sub problem
- Instant insanity when done by hand

Transformation Pipeline

1. **Desugaring**
   - comprehensions -> function calls
2. **Vectorization/Flattening**
   - maps Nested Data parallelism to Flat Data parallelism
3. **Distribution**
   - splits computation between a gang of threads
4. **Fusion**
   - eliminate intermediate arrays and unnecessary synchronization points in sequential code executed on each thread

Fusion

- Flattening is not enough

```
vecMul :: [:Float:] -> [:Float:] -> Float
vecMul v1 v2 = sumP [: f1*f2 | f1 <- v1 | f2 <- v2 :]
```

- Do not
  1. Generate [: f1*f2 | f1 <- v1 | f2 <- v2 :] (big intermediate vector)
  2. Add up the elements of this vector
- Instead: multiply and add in the same loop
- That is, **fuse** the multiply loop with the add loop
- Very general, aggressive fusion is required

Purity pays off

- Extensive source-to-source transformations
- Depend utterly on purely-functional semantics:
  - no assignments
  - every operation is a pure function

The data-parallel languages of the future will be functional languages

And it goes fast too...

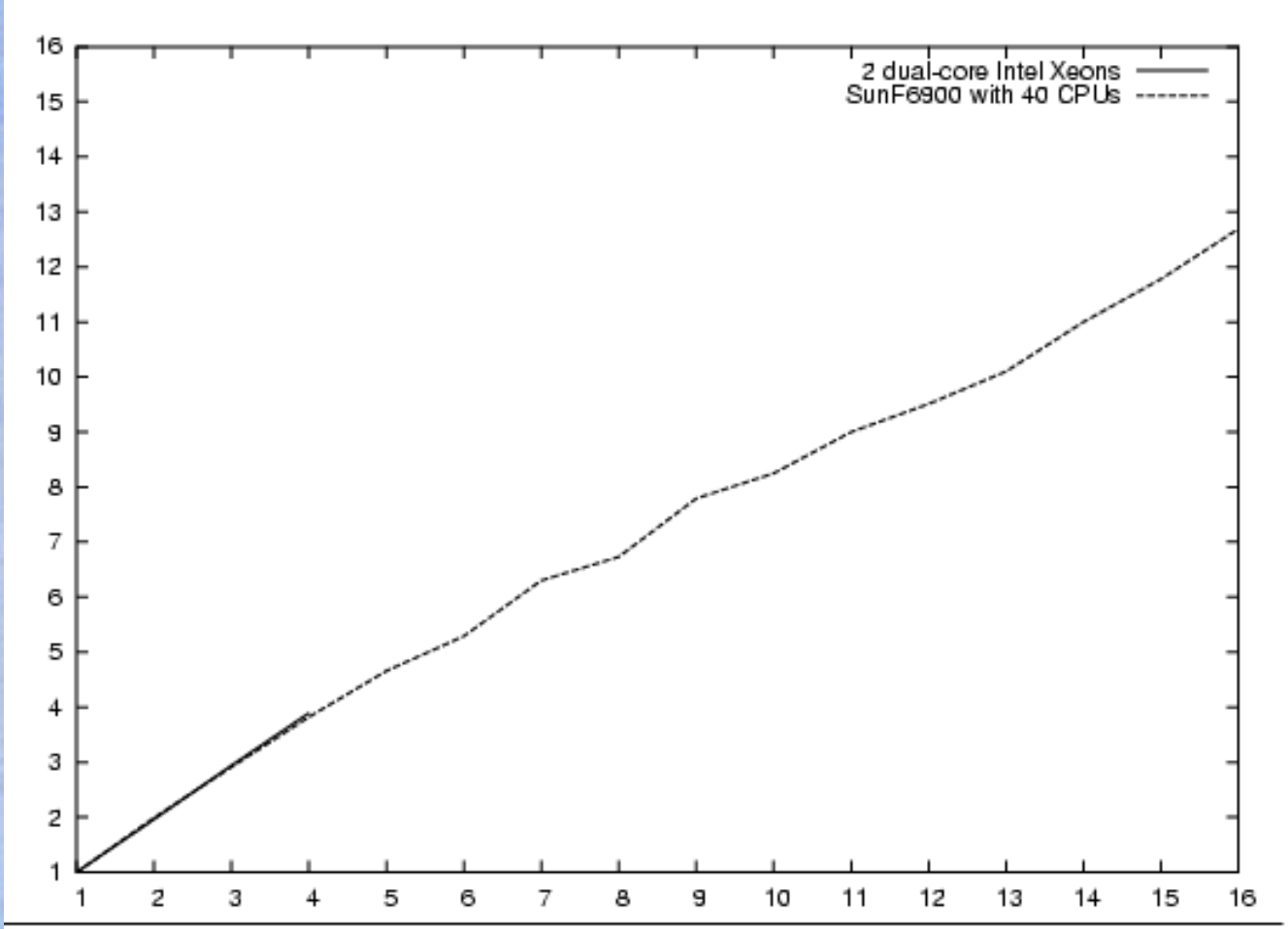


Figure 2. Speedup of smvm (x-axis is number of PEs)

1-processor version goes only 30% slower than C

Perf win with 2 processors

Summary

- Data parallelism is the only way to harness 100's of cores
- Nested DP is great for programmers: far, far more flexible than flat DP
- Nested DP is tough to implement
- But we (think we) know how to do it
- Functional programming is a massive win in this space
- Prototype in current GHC release (6.10.1)

http://haskell.org/haskellwiki/GHC/Data_Parallel_Haskell