Parallel Scanning

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

CS2101

Plan

1. Problem Statement and Applications
2. Algorithms
3. Applications
4. Implementation in Julia

Plan

Parallel scan: chapter overview

Overview

- This chapter will be the first dedicated to the applications of a parallel algorithm.
- This algorithm, called *the parallel scan*, aka *the parallel prefix sum* is a beautiful idea with surprising uses: it is a powerful recipe to turning serial into parallel.
- Watch closely what is being optimized for: this is an amazing lesson of parallelization.
- Application of parallel scan are numerous:
  - it is used in program compilation, scientific computing and,
  - we already met prefix sum with the counting-sort algorithm!

Prefix sum

Prefix sum of a vector: specification

Input: a vector $\vec{x} = (x_1, x_2, \ldots, x_n)$

Ouput: the vector $\vec{y} = (y_1, y_2, \ldots, y_n)$ such that $y_i = \sum_{i=1}^{j=i} \; x_j$ for $1 \leq j \leq n$.

Prefix sum of a vector: example

The prefix sum of $\vec{x} = (1, 2, 3, 4, 5, 6, 7, 8)$ is $\vec{y} = (1, 3, 6, 10, 15, 21, 28, 36)$.

Prefix sum: thinking of parallelization (1/2)

Remark

So a Julia implementation of the above specification would be:

```
function prefixSum(x)
   n = length(x)
   y = fill(x[1],n)
   for i=2:n
      y[i] = y[i-1] + x[i]
   end
   y
end

n = 10

x = [mod(rand(Int32),10) for i=1:n]

prefixSum(x)
```

Comments (1/2)

- The i-th iteration of the loop is not at all decoupled from the $(i-1)$-th iteration.
- Impossible to parallelize, right?

Prefix sum: thinking of parallelization (2/2)

Remark

So a Julia implementation of the above specification would be:

```
function prefixSum(x)
   n = length(x)
   y = fill(x[1],n)
   for i=2:n
      y[i] = y[i-1] + x[i]
   end
   y
end

n = 10

x = [mod(rand(Int32),10) for i=1:n]

prefixSum(x)
```

Comments (2/2)

- Consider again $\vec{x} = (1, 2, 3, 4, 5, 6, 7, 8)$ and its prefix sum $\vec{y} = (1, 3, 6, 10, 15, 21, 28, 36)$.
- Is there any value in adding, say, 4+5+6+7 on itw own?
- If we separately have 1+2+3, what can we do?
- Suppose we added 1+2, 3+4, etc. pairwise, what could we do?

Parallel scan: formal definitions

- Let S be a set, let $+ : S \times S \to S$ be an associative operation on S with 0 as identity. Let $A[1 \cdots n]$ be an array of n elements of S.
- Tthe *all-prefixes-sum* or *inclusive scan* of A computes the array B of n elements of S defined by

$$B[i] = \begin{cases} A[1] & \text{if} \quad i = 1 \\ B[i-1] + A[i] & \text{if} \quad 1 < i \leq n \end{cases}$$

- The *exclusive scan* of A computes the array B of n elements of S:

$$C[i] = \begin{cases} 0 & \text{if} \quad i = 1 \\ C[i-1] + A[i-1] & \text{if} \quad 1 < i \leq n \end{cases}$$

- An exclusive scan can be generated from an inclusive scan by shifting the resulting array right by one element and inserting the identity.
- Similarly, an inclusive scan can be generated from an exclusive scan.

Plan

Serial scan: pseudo-code

Here's a sequential algorithm for the inclusive scan.

```
function prefixSum(x)
   n = length(x)
   y = fill(x[1],n)
   for i=2:n
      y[i] = y[i-1] + x[i]
   end
   y
end
```

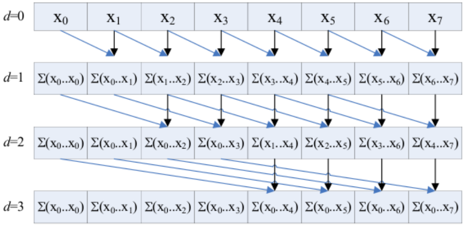
Comments

- Recall that this is similar to the *cumulated frequency computation* that is done in the prefix sum algorithm.
- Observe that this sequential algorithm performa $n-1$ additions.

Naive parallelization (1/4)

Principles

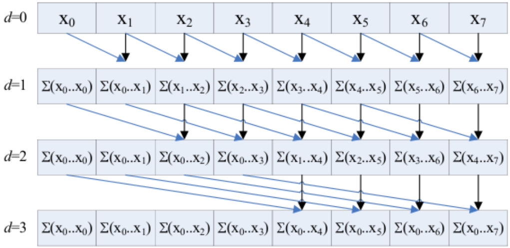
- Assume we have the input array has `n` entries and we have `n` workers at our disposal
- We aim at doing as much as possible per parallel step. For simplicity, we assume that n is a power of 2.
- Hence, during the first parallel step, each worker (except the first one) adds the value it owns to that of its left neighbour: this allows us to compute all sums of the forms $x_{k-1} + x_{k-2}$, for $2 \leq k \leq n$.
- For this to happen, we need to work OUT OF PLACE. More precisely, we need an auxiliary with `n` entries.



Naive parallelization (2/4)

Principles

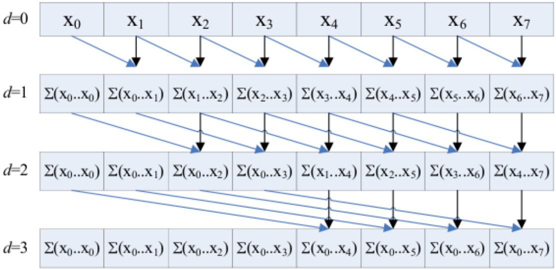
- Recall that the k-th slot, for $2 \leq k \leq n$, holds $x_{k-1} + x_{k-2}$.
- If $n = 4$, we can conclude by adding Slot 0 and Slot 2 on one hand and Slot 1 and Slot 3 on the other.
- More generally, we can perform a second parallel step by adding Slot k and Slot $k - 2$, for $3 \leq k \leq n$.



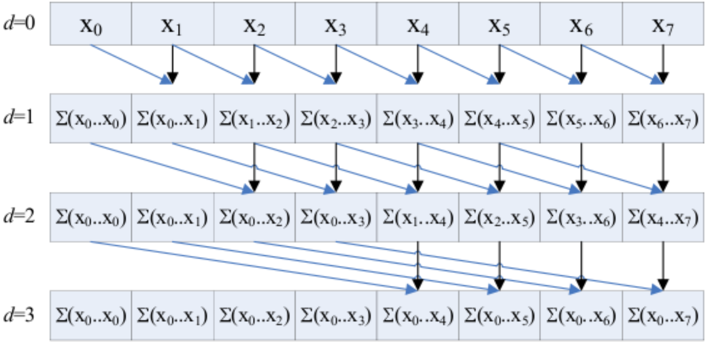
Naive parallelization (3/4)

Principles

- Now the k-th slot, for $4 \leq k \leq n$, holds $x_{k-1} + x_{k-2} + x_{k-3} + x_{k-4}$.
- If $n = 8$, we can conclude by adding Slot 5 and Slot 1, Slot 6 and Slot 2, Slot 7 and Slot 3, Slot 8 and Slot 4.
- More generally, we can perform a third parallel step by adding Slot k and Slot $k-4$ for $5 \leq k \leq n$.



Naive parallelization (4/4)



Naive parallelization: pseudo-code (1/2)

Input: Elements located in $M[1], \ldots, M[n]$, where n is a power of 2.

Output: The n prefix sums located in $M[n+1], \ldots, M[2n]$.

Program:

```
Active Proocessors P[1], ...,P[n];
// id the active processor index
for d := 0 to (log(n) -1) do
if d is even then
  if id > 2^d then
      M[n + id] := M[id] + M[id - 2^d]
  else
      M[n + id] := M[id]
  end if
else
  if id > 2^d then
      M[id] := M[n + id] + M[n + id - 2^d]
  else
      M[id] := M[n + id]
  end if
end if
if d is odd then M[n + id] := M[id] end if
```

Naive parallelization: pseudo-code (2/2)

Pseudo-code

```
Active Proocessors P[1], ...,P[n]; // id the active processor index
for d := 0 to (log(n) -1) do
if d is even then
  if id > 2^d then
      M[n + id] := M[id] + M[id - 2^d]
  else
      M[n + id] := M[id]
  end if
else
  if id > 2^d then
      M[id] := M[n + id] + M[n + id - 2^d]
  else
      M[id] := M[n + id]
  end if
end if
if d is odd then M[n + id] := M[id] end if
```

Observations

- $M[n+1], \ldots, M[2n]$ are used to hold the intermediate results at Steps $d = 0, 2, 4, \ldots (\log(n) - 2)$.
- Note that at Step d, $(n - 2^d)$ processors are performing an addition.
- Moreover, at Step d, the distance between two operands in a sum is 2^d.

Naive parallelization: analysis

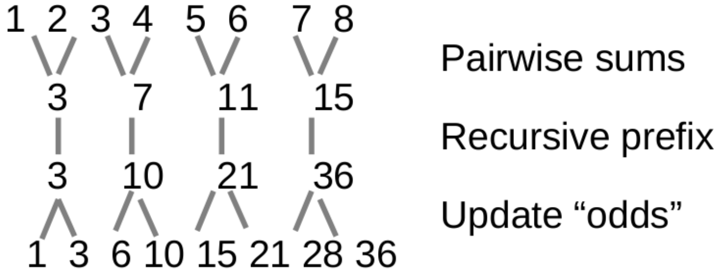
Recall

- $M[n+1], \ldots, M[2n]$ are used to hold the intermediate results at Steps $d = 0, 2, 4, \ldots (\log(n) - 2)$.
- Note that at Step d, $(n - 2^d)$ processors are performing an addition.
- Moreover, at Step d, the distance between two operands in a sum is 2^d.

Analysis

- It follows from the above that the naive parallel algorithm performs $\log(n)$ parallel steps
- Moreover, at each parallel step, at least $n/2$ additions are performed.
- Therefore, this algorithm performs at least $(n/2)\log(n)$ additions
- Thus, this algorithm is not work-efficient since the work of our serial algorithm is simply $n - 1$ additions.

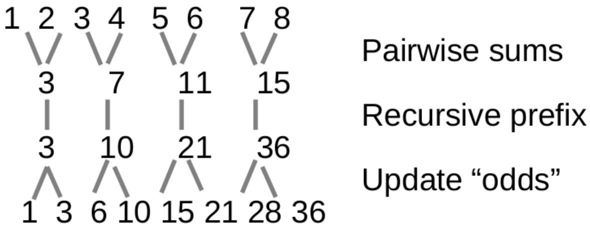
Parallel scan: a recursive work-efficient algorithm (1/2)



Algorithm

- Input: $x[1], x[2], \ldots, x[n]$ where n is a power of 2.
- Step 1: $(x[k], x[k-1]) = (x[k] + x[k-1], x[k]$ for all even k's.
- Step 2: Recursive call on $x[2], x[4], \ldots, x[n]$
- Step 3: $x[k-1] = x[k] - x[k-1]$ for all even k's.

Parallel scan: a recursive work-efficient algorithm (2/2)



Analysis

- Since the recursive call is applied to an array of size $n/2$, the total number of recursive calls is $\log(n)$.
- Before the recursive call, one performs $n/2$ additions
- After the recursive call, one performs $n/2$ subtractions
- Elementary calculations show that this recursive algorithm performs at most a total of $2n$ additions and subtractions
- Thus, this algorithm is work-efficient. In addition, it can run in $2\log(n)$ parallel steps.

Plan

Application to Fibonacci sequence computation

$$\boxed{F_{n+1} = F_n + F_{n-1}}$$

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

Can compute all F_n by matmul_prefix on

$$\left[\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \right]$$

Application to parallel addition (1/2)

Example						Notation			
1	0	1	1	1		Carry	c_2	c_1	c_0	
	1	0	1	1	1	First Int	a_3	a_2	a_1	a_0
	1	0	1	0	1	Second Int	b_3	b_2	b_1	b_0

Application to parallel addition (2/2)

Example						
1	0	1	1	1		Carry
	1	0	1	1	1	First Int
	1	0	1	0	1	Second Int

Notation			
c_2	c_1	c_0	
a_3	a_2	a_1	a_0
a_3	b_2	b_1	b_0

$c_{-1} = 0$

(addition mod 2)

for $i = 0 : n-1$

$s_i = a_i + b_i + c_{i-1}$

$c_i = a_i b_i + c_{i-1}(a_i + b_i)$

end

$$\begin{bmatrix} c_i \\ 1 \end{bmatrix} = \begin{bmatrix} a_i + b_i & a_i b_i \\ 0 & 1 \end{bmatrix} \begin{bmatrix} c_{i-1} \\ 1 \end{bmatrix}$$

Plan

Serial prefix sum: recall

```
function prefixSum(x)
   n = length(x)
   y = fill(x[1],n)
   for i=2:n
      y[i] = y[i-1] + x[i]
   end
   y
end

n = 10

x = [mod(rand(Int32),10) for i=1:n]

prefixSum(x)
```

Parallel prefix multiplication: live demo (1/7)

```
julia> reduce(+,1:8) #sum(1:8)
36


julia> reduce(*, 1:8) #prod(1:8)
40320

julia> boring(a,b)=a
# methods for generic function boring
boring(a,b) at none:1

julia> println(reduce(boring, 1:8))
1

julia> boring2(a,b)=b
# methods for generic function boring2
boring2(a,b) at none:1

julia> reduce(boring2, 1:8)
8
```

Comments

- First, we test Julia's `reduce` function with different operations.

Parallel prefix multiplication: live demo (2/7)

```
julia> fib(j)=reduce(*, [[[1, 1] [1, 0]] for i=1:j])
# methods for generic function fib
fib(j) at none:1

julia> map(fib, [4, 7])
2-element Array{Array{Int64,2},1}:
 2x2 Array{Int64,2}:
 5  3
 3  2
 2x2 Array{Int64,2}:
 21  13
 13   8

julia> Hadamard(n)=reduce(kron, [[[1,1] [1,-1]] for i=1:n])
# methods for generic function Hadamard
Hadamard(n) at none:1

julia> Hadamard(3)
8x8 Array{Int64,2}:
 1   1   1   1   1   1   1   1
 1  -1   1  -1   1  -1   1  -1
 1   1  -1  -1   1   1  -1  -1
 1  -1  -1   1   1  -1  -1   1
 1   1   1   1  -1  -1  -1  -1
 1  -1   1  -1  -1   1  -1   1
 1   1  -1  -1  -1  -1   1   1
 1  -1  -1   1  -1   1   1  -1
```

Comments

- Next, we compute Fibonacci numbers and Hadamard matrices via prefix sum.

Parallel prefix multiplication: live demo (3/7)

```
julia> M=[randn(2,2) for i=1:4];

julia> printnice(x)=println(round(x,3))
# methods for generic function printnice
printnice(x) at none:1

julia> printnice(M[4]*M[3]*M[2]*M[1])
-.466 .906
1.559 -3.447


julia> printnice(reduce((A,B)->B*A, M)) #backward multiply
-.466 .906
1.559 -3.447


julia> printnice(reduce(*, M))          #forward multiply
-.823 .25
-2.068 .39
```

Comments

- In the above we do a prefix multiplication with random matrices.

Parallel prefix multiplication: live demo (4/7)

```
julia> h=reduce((f,g)->(x->f(g(x))), [sin cos tan])
# function

julia>

julia> [h(pi) sin(cos(tan(pi)))]
1x2 Array{Float64,2}:
 0.841471  0.841471
```

Comments

- In the above example we apply 'reduce()' to function composition:

Parallel prefix multiplication: live demo (5/7)

```
julia> @everywhere function prefix_serial!(y,*)
    @inbounds for i in 2:length(y)
        y[i]=y[i-1]*y[i]
    end
    y
end;

julia> function prefix8!(y,*)
           if length(y)!=8; error("length 8 only"); end
           for i in [2,4,6,8]; y[i]=y[i-1]*y[i]; end
           for i in [  4,  8]; y[i]=y[i-2]*y[i]; end
           for i in [      8]; y[i]=y[i-4]*y[i]; end
           for i in [    6  ]; y[i]=y[i-2]*y[i]; end
           for i in [ 3,5,7 ]; y[i]=y[i-1]*y[i]; end
           y
       end
# methods for generic function prefix8!
prefix8!(y,*) at none:2

julia> function prefix!(y,.*)
           l=length(y)
           k=int(ceil(log2(l)))
           @inbounds for j=1:k, i=2^j:2^j:min(l, 2^k)              #"reduce"
               y[i]=y[i-2^(j-1)].*y[i]
           end
           @inbounds for j=(k-1):-1:1, i=3*2^(j-1):2^j:min(l, 2^k) #"broadcast"
               y[i]=y[i-2^(j-1)].*y[i]
           end
           y
       end
# methods for generic function prefix!
prefix!(y,.*) at none:2
```

Comments

- We prepare a prefix-sum computation with 8 workers and 8 matrices to multiply.

Parallel prefix multiplication: live demo (6/7)

```
+(r1::RemoteRef,r2::RemoteRef)=@spawnat r2.where fetch(r1)+fetch(r2)

 methods for generic function +
+(x::Bool,y::Bool) at bool.jl:38
+(x::Int64,y::Int64) at int.jl:36

... 91 methods not shown (use methods(+) to see them all)

julia> *(r1::RemoteRef,r2::RemoteRef)=@spawnat r2.where fetch(r1)*fetch(r2)
# methods for generic function *

... 121 methods not shown (use methods(*) to see them all)


julia> # The serial version requires 7 operations. The parallel version uses
```

Comments

- We prepare a prefix-sum computation with 8 workers and 8 matrices to multiply.

Parallel prefix multiplication: live demo (7/7)

```
\julia> n=2048
2048

julia> r=[@spawnat i randn(n,n) for i=1:8]; s=fetch(r); t=copy(r)
8-element Array{Any,1}:
 RemoteRef(1,1,16)
 RemoteRef(2,1,17)
 RemoteRef(3,1,18)
 RemoteRef(4,1,19)
 RemoteRef(5,1,20)
 RemoteRef(6,1,21)
 RemoteRef(7,1,22)
 RemoteRef(8,1,23)

julia> tic(); prefix_serial!(s, *); t_ser = toc()
elapsed time: 10.679596478 seconds
10.679596478

julia> tic(); @sync prefix8!(t, *); t_par = toc() #Caution: race condition bug #4330
elapsed time: 7.434856303 seconds
7.434856303

julia> @printf("Serial: %.3f sec  Parallel: %.3f sec  speedup: %.3fx (theory=1.4x)", t_ser, t_par, t_ser/
Serial: 10.680 sec  Parallel: 7.435 sec  speedup: 1.436x (theory=1.4x)
```

Comments

- Now let's run prefix in parallel on 8 processors.