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

1. Problem Statement and Applications
2. Algorithms
3. Applications
4. Implementation in Julia
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 of a vector: specification

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

Output: the vector $\vec{y} = (y_1, y_2, \ldots, y_n)$ such that $y_i = \sum_{j=1}^{i} x_j$ 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 $\vec{y} = (1, 3, 6, 10, 15, 21, 28, 36)$. 
Remark
So a Julia implementation of the above specification would be:

```julia
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
```

```julia
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:

```julia
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 \( \bar{x} = (1, 2, 3, 4, 5, 6, 7, 8) \) and its prefix sum \( \bar{y} = (1, 3, 6, 10, 15, 21, 28, 36) \).
- Is there any value in adding, say, 4+5+6+7 on its 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?
Let $S$ be a set, let $+: S \times S \rightarrow S$ be an associative operation on $S$ with $0$ as identity. Let $A[1 \cdots n]$ be an array of $n$ elements of $S$.

The all-prefixes-sum or inclusive scan of $A$ computes the array $B$ of $n$ elements of $S$ defined by

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

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

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

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

1. Problem Statement and Applications
2. Algorithms
3. Applications
4. Implementation in Julia
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.
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 $3 \leq k \leq n$. 

```
\begin{array}{c|c|c|c|c|c|c|c|c|c}
 & X_0 & X_1 & X_2 & X_3 & X_4 & X_5 & X_6 & X_7 \\
\hline
d=0 & \Sigma(x_0..x_0) & \Sigma(x_0..x_1) & \Sigma(x_0..x_2) & \Sigma(x_0..x_3) & \Sigma(x_0..x_4) & \Sigma(x_0..x_5) & \Sigma(x_0..x_6) & \Sigma(x_0..x_7) \\
\hline
d=1 & \Sigma(x_0..x_0) & \Sigma(x_0..x_1) & \Sigma(x_0..x_2) & \Sigma(x_0..x_3) & \Sigma(x_0..x_4) & \Sigma(x_0..x_5) & \Sigma(x_0..x_6) & \Sigma(x_0..x_7) \\
\hline
d=2 & \Sigma(x_0..x_0) & \Sigma(x_0..x_1) & \Sigma(x_0..x_2) & \Sigma(x_0..x_3) & \Sigma(x_0..x_4) & \Sigma(x_0..x_5) & \Sigma(x_0..x_6) & \Sigma(x_0..x_7) \\
\hline
d=3 & \Sigma(x_0..x_0) & \Sigma(x_0..x_1) & \Sigma(x_0..x_2) & \Sigma(x_0..x_3) & \Sigma(x_0..x_4) & \Sigma(x_0..x_5) & \Sigma(x_0..x_6) & \Sigma(x_0..x_7) \\
\end{array}
```
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 $5 \leq k \leq n$. 

![Diagram of parallel steps](image)
### Naive parallelization (4/4)

<table>
<thead>
<tr>
<th>$d=0$</th>
<th>$X_0$</th>
<th>$X_1$</th>
<th>$X_2$</th>
<th>$X_3$</th>
<th>$X_4$</th>
<th>$X_5$</th>
<th>$X_6$</th>
<th>$X_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\Sigma(x_0 \ldots x_0)$</td>
<td>$\Sigma(x_0 \ldots x_1)$</td>
<td>$\Sigma(x_0 \ldots x_2)$</td>
<td>$\Sigma(x_1 \ldots x_3)$</td>
<td>$\Sigma(x_2 \ldots x_4)$</td>
<td>$\Sigma(x_3 \ldots x_5)$</td>
<td>$\Sigma(x_4 \ldots x_6)$</td>
<td>$\Sigma(x_6 \ldots x_7)$</td>
</tr>
<tr>
<td>$d=1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\Sigma(x_0 \ldots x_0)$</td>
<td>$\Sigma(x_0 \ldots x_1)$</td>
<td>$\Sigma(x_0 \ldots x_2)$</td>
<td>$\Sigma(x_0 \ldots x_3)$</td>
<td>$\Sigma(x_1 \ldots x_4)$</td>
<td>$\Sigma(x_2 \ldots x_5)$</td>
<td>$\Sigma(x_3 \ldots x_6)$</td>
<td>$\Sigma(x_4 \ldots x_7)$</td>
</tr>
<tr>
<td>$d=2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\Sigma(x_0 \ldots x_0)$</td>
<td>$\Sigma(x_0 \ldots x_1)$</td>
<td>$\Sigma(x_0 \ldots x_2)$</td>
<td>$\Sigma(x_0 \ldots x_3)$</td>
<td>$\Sigma(x_0 \ldots x_4)$</td>
<td>$\Sigma(x_0 \ldots x_5)$</td>
<td>$\Sigma(x_0 \ldots x_6)$</td>
<td>$\Sigma(x_0 \ldots x_7)$</td>
</tr>
<tr>
<td>$d=3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 Procecssors $P[1], \ldots, P[n]$;

// 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 Processors 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.

**Diagram:**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>10</td>
<td>21</td>
<td>36</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>21</td>
<td>28</td>
<td>36</td>
</tr>
</tbody>
</table>

Pairwise sums
Recursive prefix
Update “odds”
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

1 Problem Statement and Applications
2 Algorithms
3 Applications
4 Implementation in Julia
Application to Fibonacci sequence computation

\[
F_{n+1} = F_n + F_{n-1}
\]

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

Can compute all \( F_n \) by `matmul_prefix` on

\[
\left[
\begin{array}{cccccccc}
1 & 1 \\
1 & 0
\end{array},
\begin{array}{cccccccc}
1 & 1 \\
1 & 0
\end{array},
\begin{array}{cccccccc}
1 & 1 \\
1 & 0
\end{array},
\begin{array}{cccccccc}
1 & 1 \\
1 & 0
\end{array},
\begin{array}{cccccccc}
1 & 1 \\
1 & 0
\end{array},
\begin{array}{cccccccc}
1 & 1 \\
1 & 0
\end{array},
\begin{array}{cccccccc}
1 & 1 \\
1 & 0
\end{array},
\begin{array}{cccccccc}
1 & 1 \\
1 & 0
\end{array}
\right]
\]
## Application to parallel addition (1/2)

<table>
<thead>
<tr>
<th>Example</th>
<th>Carry</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 1</td>
<td>1 0 1 1 1</td>
<td>(c_2\ c_1\ c_0)</td>
</tr>
<tr>
<td>1 0 1 0 1</td>
<td>First Int</td>
<td>(a_3\ a_2\ a_1\ a_0)</td>
</tr>
<tr>
<td></td>
<td>Second Int</td>
<td>(b_3\ b_2\ b_1\ b_0)</td>
</tr>
</tbody>
</table>
### Application to parallel addition (2/2)

<table>
<thead>
<tr>
<th>Example</th>
<th>Carry</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 1 1</td>
<td><strong>First Int</strong></td>
<td>( a_3 \ a_2 \ a_1 \ a_0 )</td>
</tr>
<tr>
<td>1 0 1 0 1</td>
<td><strong>Second Int</strong></td>
<td>( a_3 \ b_2 \ b_1 \ b_0 )</td>
</tr>
</tbody>
</table>

\[ c_{-1} = 0 \]

(addition mod 2)

For \( i = 0 : n-1 \)

\[ s_i = a_i + b_i + c_{i-1} \]

\[ c_i = a_i b_i + c_{i-1}(a_i + b_i) \]

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

end
Plan

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

julia> n = 3000; k = 3;

julia> v=[randn(n,n) for i=1:2^k];

julia> w=copy(v);

julia> @time for i=2:2^k
    w[i]=w[i-1]*v[i];
    end

elapsed time: 32.458615523 seconds (516419092 bytes allocated)

Comments

- In the above we do a prefix multiplication with random matrices.
- We have $n = 2^k$.
- After randomly generating the matrices, we do the serial prefix mult.
Parallel prefix multiplication: live demo (2/4)

julia> l
4

julia> k
3

julia> p=workers()
4-element Array{Int64,1}:
  2
  3
  4
  5

julia> l=length(p)
4

julia> if 1<2^k;
    addprocs(2^k-1+(l==1));
    p=workers();
end
8-element Array{Int64,1}:
  2
  3
  4
  5
  6
  7
  8
  9

Comments
- We enforce $2^k$ worker processors.
Parallel prefix multiplication: live demo (3/4)

```julia
r=@spawnat p[i] randn(n,n) for i=1:2^k
```

8-element Array{Any,1}:
  RemoteRef(2,1,1)
  RemoteRef(3,1,2)
  RemoteRef(4,1,3)
  RemoteRef(5,1,4)
  RemoteRef(6,1,5)
  RemoteRef(7,1,6)
  RemoteRef(8,1,7)
  RemoteRef(9,1,8)

```
julia> s=copy(r)
```

8-element Array{Any,1}:
  RemoteRef(2,1,1)
  RemoteRef(3,1,2)
  RemoteRef(4,1,3)
  RemoteRef(5,1,4)
  RemoteRef(6,1,5)
  RemoteRef(7,1,6)
  RemoteRef(8,1,7)
  RemoteRef(9,1,8)

Comments

- We create remote random matrices.
Parallel prefix multiplication: live demo (4/4)

@time @sync begin
    for j=1:k
        for i in [2^j:2^j:2^k]
            s[i]=@spawnat p[i] fetch(s[i-2^(j-1)])*fetch(s[i]);
        end
    end
    for j=(k-1):-1:1
        for i in [3*2^(j-1):2^j:2^k]
            s[i]=@spawnat p[i] fetch(s[i-2^(j-1)])*fetch(s[i]);
        end
    end
end

elapsed time: 20.513351976 seconds (5045520 bytes allocated)

Comments

•

•

•