Example: Prefix Sum

Recursive Doubling with Barriers

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

© 2018 L. V. Kale at the University of Illinois Urbana
Prefix Sum Problem

- Given array $A[0..N-1]$, produce $B[N]$, such that $B[k]$ is the sum of all elements of $A$ up to $A[k]$.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

Prefix Sum: A good Sequential Algorithm

• Data dependency from iteration to iteration
  • How can this be parallelized at all?

\[
\begin{align*}
B[0] &= A[0]; \\
\text{for } (i=1; i<N; i++) &\quad B[i] = B[i-1] + A[i];
\end{align*}
\]

• It looks like the problem is inherently sequential, but theoreticians came up with a beautiful algorithm called recursive doubling or just parallel prefix
Parallel Prefix: Recursive Doubling

N Data Items
P Processors
N=P

Log P Phases
P additions in each phase
P log P operations
Completes in $O(\log P)$ time

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>7</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>15</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OpenMP Formulation for Parallel Prefix

- We don’t have \( n \) processors
  - I.e., the number of threads will be much smaller than the size of the data array
- So, we will simulate \( n \) processors using \( p \) threads
- Notice that each phase of the computation must finish before the next phase of the computation starts
  - We will use OpenMP’s \texttt{barrier} directive for that
- Basic description of actions in each phase with distance \( d \)
  - Every “processor” \( i \) adds its value to the value held by a processor distance \( d \) away
  - Simulation: \( B[i+d] \texttt{+=} B[i] \), but you have to be careful to avoid dependencies
    - I.e., copy \( B[i] \) into a temporary variable at \( i+d \), say \( C[i+d] \) and then add \( C[i] \) to \( B[i] \) for every \( i \)
  - Note \( d \) doubles in every phase
Parallel Prefix: Recursive Doubling

0 1 2 3 4 5 6 7

5 3 7 2 1 3 1 2

- - - - - - - -

5 3 7 2 1 3 1

5

B

C

B

L.V.Kale
#pragma omp parallel for
for(i=0;i<n;i++){B[i]=A[i];}
int d=1;
while(d<n) // this loop will run for lg n steps
{
    int i;
    #pragma omp parallel for
    for(i=d;i<n;i++)C[i]=B[i-d];

    #pragma omp parallel for
    for(i=d;i<n;i++)B[i]+=C[i];

d*=2;
}
Critique of Prefix Algorithm 1

• The sequential algorithm had $n$ additions
• But the parallel algorithm is doing a total of $n*(\log n)$ additions
  • Although they are parallelized by $p$ threads
  • This is an example of an algorithm that is not “work efficient”
• It uses $\log n$ barriers, which are expensive operations
• Maybe a thread oriented approach will avoid the $\log n$ factors
Prefix Sum Algorithm 2: A Thread Oriented Approach

• What if we let each thread calculate prefix sum over its own range of array?
  • I.e., thread id is responsible for range \( B_{\frac{n\cdot id}{p}}^{\frac{n\cdot (id+1)}{p} - 1} \)
  • Id : my thread’s serial number; \( p \) : total number of threads
  • Assuming \( n \) is a multiple of \( p \)

• But then each thread needs the sum of all numbers to its left
Thread\textsubscript{0} \quad \ldots \quad \text{Thread}_{id-1} \quad \ldots \quad \text{Thread}_{id} \quad \text{myBegin} \quad 
Thread_{id} \quad \text{myEnd}
Prefix Sum Algorithm 2: A Thread Oriented Approach

• What if we let each thread calculate prefix sum over its own range of array?
  • I.e., thread \(i\) is responsible for range \(B_{\frac{n \times id}{p} : \frac{n \times (id+1)}{p} - 1}\)
    
    *id – my thread’s serial number; *p – total number of threads
    
    Assuming \(n\) is a multiple of \(p\)

• But then each thread needs the sum of all numbers to its left

• If we are willing to double the amount of work, we can obtain this sum with a much smaller prefix sum problem of size \(p\)
  
  1. First loop: every thread calculates sum \(s\) over its sub-range and copies \(s\) into a shared array called \(\text{sums}[id]\)
  
  2. Calculate prefix sum of the \(\text{sums}\) array
    
    • \(\text{sums}[id-1]\) has the sum of all values to the left of thread numbered \(id\)

  3. Second loop: every thread with serial number \(id\) calculates the prefix sum in array \(B\) using \(\text{sums}[id-1]\) and the values in \(A\)
Thread_0

Thread_{id-1}

Thread_id

sums

sum

sums[\text{id-1}]

\text{myBegin}

\text{myEnd}
omp_set_num_threads(p);

#pragma omp parallel
{
    int id=omp_get_thread_num();
    int myBegin = (n*id)/p;
    int myEnd = min( (n*id+1)/p, n);

    int sum=0;
    for(int i=myBegin;i<myEnd;i++)
        sum+=B[i];
    sums[id]=sum;

#pragma omp barrier
#pragma omp single
{
    for(int i=1;i<p;i++)
        sums[i]+=sums[i-1];
}

if(id>0)B[myBegin]+= sums[id-1]
for(int i=myBegin+1;i<myEnd/p;i++)
    B[i]+=B[i-1];
}

int sum=0;
for(int i=myBegin;i<myEnd;i++)
    sum+=B[i];
sums[id]=sum;

#pragma omp barrier
#pragma omp single
{
    for(int i=1;i<p;i++)
        sums[i]+=sums[i-1];
}

if(id>0)B[myBegin]+= sums[id-1]
for(int i=myBegin+1;i<myEnd/p;i++)
    B[i]+=B[i-1];
}