Example: Prefix Sum

Recursive Doubling with Barriers: Algorithm 2
Prefix Sum: A good Sequential Algorithm

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

\[
B[0] = A[0];
\]
\[
\text{for } (i=1; i<N; i++)
B[i] = B[i-1] + A[i];
\]

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

L.V.Kale
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>
... 

```c
#pragma omp parallel for
for(i=0;i<n;i++) B[i]=A[i];
```

```c
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 $n \cdot (\log n)$ 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 $B\left[\frac{n\cdot id}{p} : \frac{n\cdot (id+1)}{p} - 1\right]$
  • 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
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 $B\left[\frac{n \cdot \text{id}}{p} : \frac{n \cdot (\text{id} + 1)}{p} - 1\right]$
    *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 sums at \text{sums}[\text{id}]
  2. Calculate prefix sum of the sums array
    • \text{sums}[\text{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}[\text{id}-1] and the values in A
Thread$_0$  

Thread$_{id-1}$  

Thread$_{id}$  

sums  

sum  

sums$_{[id-1]}$  

myBegin  

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];
}

Form Local sum
- Calculate Prefix sum of size p
- Sums [id] now contains the sum of values of all previous threads’ ranges
- This can be done in parallel but it’s not worth it

Complete the Prefix sum