Spring 2014, CSE 392, Homework 2

Abhishek Bhaduri and Aaron Stacy

1. If we expand out the first several terms of the operation we get:

$$\begin{aligned}
x_0 &= x_0 \\
x_1 &= a_1x_0 + b_1 \\
x_2 &= a_2a_1x_0 + a_2b_1 + b_2 \\
x_3 &= a_3a_2a_1x_0 + a_3a_2b_1 + a_3b_2 + b_3 \\
x_4 &= a_4a_3a_2a_1x_0 + a_4a_3a_2b_1 + a_4a_3b_2 + a_4b_3 + b_4
\end{aligned}$$

   If we define c_i as x_0 times the prefix product of the a_i values (letting $a_0 = 1$), we get:

$$\begin{aligned}
x_0 &= c_0 \\
x_1 &= c_1 + b_1 \\
x_2 &= c_2 + \frac{c_2}{c_1}b_1 + b_2 \\
x_3 &= c_3 + \frac{c_3}{c_1}b_1 + \frac{c_3}{c_2}b_2 + b_3 \\
x_4 &= c_4 + \frac{c_4}{c_1}b_1 + \frac{c_4}{c_2}b_2 + \frac{c_4}{c_3}b_3 + b_4
\end{aligned}$$

   And factor out the c_i value:

$$\begin{aligned}
x_0 &= c_0\left(\frac{1}{1}\right) \\
x_1 &= c_1\left(\frac{1}{1} + \frac{b_1}{c_1}\right) \\
x_2 &= c_2\left(\frac{1}{1} + \frac{b_1}{c_1} + \frac{b_2}{c_2}\right) \\
x_3 &= c_3\left(\frac{1}{1} + \frac{b_1}{c_1} + \frac{b_2}{c_2} + \frac{b_3}{c_3}\right) \\
x_4 &= c_4\left(\frac{1}{1} + \frac{b_1}{c_1} + \frac{b_2}{c_2} + \frac{b_3}{c_3} + \frac{b_4}{c_4}\right)
\end{aligned}$$

   So it's clear that if we define $d_i = \frac{b_i}{c_i}$, and set e_i to the prefix sum of d_i, we can accomplish this by doing two scans (one for c_i and one for e_i), and then setting $x_i = c_id_i$. Work-depth pseudocode:

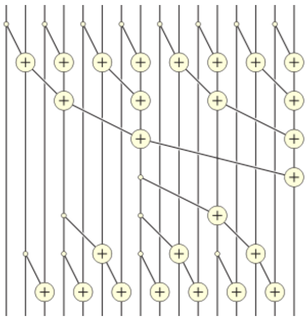
```
function operation(a, b, n)
% assume 'multiply' and 'add' are associative operations and 'scan' is a
% parallel scan function like the one from class
c = scan(a, multiply)
parfor i=0:n-1
  d(i) = b(i) / c(i)
end
e = scan(d, add)
parfor i=0:n-1
  x(i) = c(i) * e(i)
end
return e
```

2. This is implemented in [`src/scan.cc`](src/scan.cc). A few comments:

   - In the interest of portability, the signature is modified to accept the operation as a function-pointer argument.
   - The program interface accepts a whitespace-separated sequence of ascii numbers on `stdin`, and outputs the result of the scan as a whitespace-separated sequence of ascii numbers on `stdout`. Numbers are always assumed to be `double`'s. Timing is reported on file descriptor 3 if it is open when the program executes.
   - The program accepts command line arguments:

    - `-d`: Set the dimensionality of the input (i.e. 1-D vs. 4-D array elements)
    - `-m`: Generate an array of mock input data for performance testing
    - `-n`: Do not output the result (also for performance testing)

  - The algorithm is similar to the one given on slide 7 of lecture 6, except it does the calculations in place, similar to the diagram for the Wikipedia page for prefix sum:

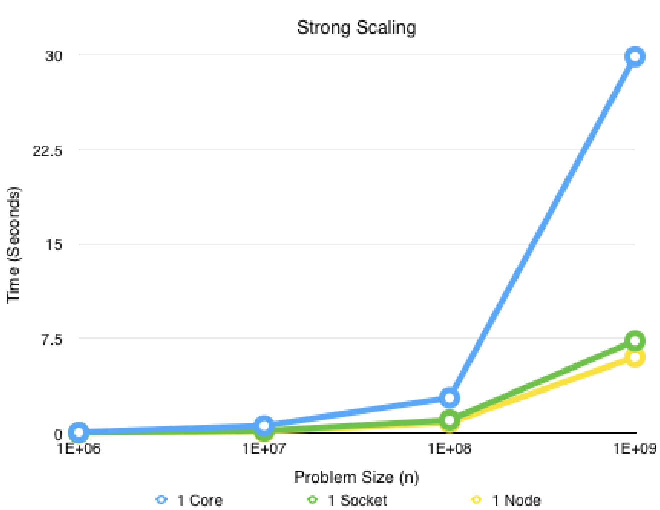


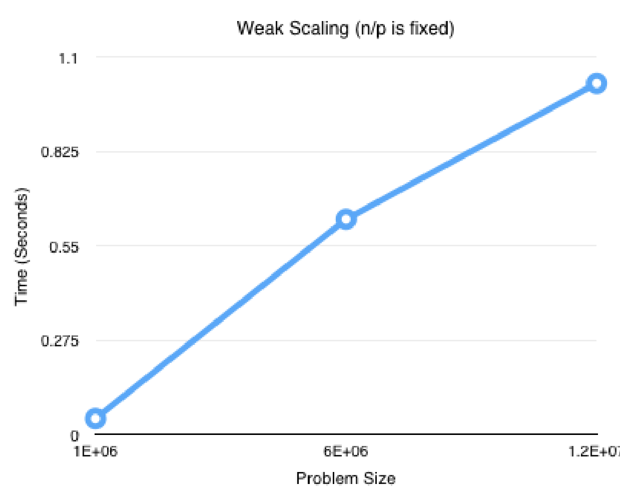
It uses a `stride` variable to track the distance between the elements being added. Pseudo-code for the case where the operation is addition is below:

```
function rec_scan(a, n, stride)
if (stride >= n); return; end;
parfor i=stride-1:n-1:i+=stride*2
  a(i+stride) = a(i) + a(i+stride)
end
rec_scan(a, n, stride*2)
parfor i=stride-1:n-stride:i+=stride
  a(i+stride/2) = a(i) + a(i+stride/2)
end
```

Results for time in seconds are below.

Problem Size	Single Core	Single Socket	Single Node
1m	0.047790	0.014720	0.019102
10m	0.580920	0.166679	0.117725
100m	2.790022	1.029475	0.823568
300m	9.211940	3.697637	2.951852
1b	29.877337	7.312807	6.025247
300m 4D	22.078455	8.007848	7.067054





3. OMP algorithm:

```
ompBSearch(....) {
//data size: n
//# threads: t
//key, arr[], id (0 to t-1)
size=ceil(n/t)
seg_start=id*size
// data size is less than number of threads available seg_end=(id+1)*size-1
if (seg_start>=n) return
// resolve boundary conditions
if (seg_end>=n || (id==t-1 && seg_end<n-1)) {
seg_end=n-1
size=seg_end-seg_start+1 }
// key is not in this block //search now
if (key<seg_start || key>seg_end) return NULL
seg_center=seg_start+size/2
// key not found if (seg_start <= key < seg_center)
if (seg_start==seg_end && seg_center != key) return NULL
ompBsearch(seg_start, size/2, key) else if (seg_center < key <= seg_finish)
ompBsearch(seg_center+1, size-size/2-1, key) else
return seg_center // Key found }
```

MPI algorithm

mpiBsearch(....) {
// num procs = p
split array of size n in p chunks of n/p
MPI_Scatter each chunk to the procs in the group
Each processor runs ompBSearch(....) (shown above) and reports
- either NULL for key not found
- or the element that matched (only one because the array has no duplicates) Report result
}
The implementation is not complete and no performance numbers are available to report. The working code so far developed is included. It is the OMP part. The MPI code is not included because it is work in progress.

The command accepts:
-n for array size
-k for key value (optional if not mentioned then a random key is used)
The program generates a sorted array of random unsigned integers. The search is applied on this array.

4. PRAM pseudocode:

```
function c = parallel_merge(a, n, b, m, p, tid)
% allocate space for result and splitters. could optimize by assigning
% each allocation to a different thread
% assume merge() is a sequential merge
if tid == 0
  c = zeros(n + m)
  aSplitters = zeros(n/p)
  bSplitters = zeros(n/p)
end

% part 1: ranking, O(logn)
as = rank(a(tid*n/p), b) % CW
bs = rank(b(tid*m/p), a) % CW
aSplitters(tid) = as
bSplitters(tid) = bs

% part 2: parallel merge, O(n/p)
if tid == 0
  c(0:as+bs) = merge(a(0, as), as, b(0, bs), bs)
else
  prevAs = aSplitters(tid - 1)
  prevBs = bSplitters(tid - 1)
  c(prevAs+prevBs:as+bs) = merge(a(prevAs, as), as-prevAs,
                                 b(prevBs, bs), bs-prevBs)
end
if tid == p-1
  c(as+bs:m+n) = merge(a(as:n), n-as, b(bs:m), m-bs)
end
```

5. A parallel version of the quickselect algorithm:

```
function m = parallel_median(a, n, below=0, above=0)
% this is the count of array items below and above the current
% partition. the `below` and `above` parameters are just used when
% recursing to keep track of where the current partition is in the
% array
pivot = a(n/2)
greater_than_pivot = zeros(n-below-above) % allocation O(n)
parfor i = below:n-above
  greater_than_pivot(i) = a(i) < n
end
count = parallel_sum(a, n) % reduction w/ addition
if below + count == n/2
  m = a(count)
else if below + count > n/2
  m = parallel_median(a, n, below, above+n-count)
else
  m = parallel_median(a, n, below+n-count, above)
end
```

Similarly to sequential versions of quicksort and quickselect, the average running time is much better than the worst-case, so we've reported θ times below instead of upper bounds.

$$
\begin{aligned}
W(n) &= \theta\left(n + \log n + W\left(\frac{n}{2}\right)\right) \\
&= \theta(n) \\
D(n) &= \theta\left(\log n + D\left(\frac{n}{2}\right)\right) \\
&= \theta(\log n)
\end{aligned}
$$

This algorithm is work-optimal in the average case.