CS 525 Homework 2

Junchao Yan
yan114@purdue.edu

February 12, 2014

1 Computing the Pagerank vector

1. *Write down the global-address space parallel algorithm for computing the Pagerank vector using the algorithmic notation we have used in class.*

   Function Pagerank

```
Input: n, rbegin, colindex, value, threshold, a
Output: x

while (max_error > threshold) do {

        max_error = 0;

#pragma omp parallel shared(a,n) reduction(max:max_error)
{

        int k,k1,k2,j;                          //counter, index
        double prev,temp,error;

#pragma omp for
        for row i = 0 to n-1 do
                temp = 0;
                k1 = rbegin[i];
                k2 = rbegin[i+1] - 1;
                if k2 < k1 then continue; end if
                for k = k1 to k2 do
                        j = coloindex[k];
                        temp += value[k]*x[j];
                end for
                temp = a*temp + (1-a)/n;  //a is the alpha
                prev = x[i];                 // prev is used to store the previous x_i
                y[i] = temp;               // y is a vector to store the updated x_i
                error = fabs(temp - prev);
```

```
            if (error > max_error) then max_error = error;
        end for

#pragma omp for
        for row i = 0 to n-1
            x[i] = y[i];
        end for

} // end parallel
} // end while

return x;
```

2. *Analyze the parallel time complexity of one iteration of the PageRank computation with the two different schedules as described above.*

   There are two *for* loops in the algorithm. One is for sparse matrix multiplication; another is for updating the pagerank vector. The first loop ends with an implicit synchronization. So there are actually two parallel regions. Here, we only use dynamic scheduling for the first region since the second one is for updating the vector.

   Static

   $\lceil n/p \rceil (2\Delta + 6)$

   Dynamic

   $2 \lceil nnz(A)/p \rceil + 6 \lceil n/p \rceil$

2 Implementation

1. *Run your code on 1, 2, 4, and 8 threads on the mc machines. Report the run times and speed-ups for the three data matrices provided.*

   Dynamic scheduling

   Dataset 1
   # of iterations = 12
   chunksize: 10

   Dataset 2
   # of iterations = 42
   chunksize: 1000

Thread	Time	Speedup
1	0.02725	1
2	0.01616	1.6862623762
4	0.00892	3.0549327354
8	0.00522	5.2203065134

Thread	Time	Speedup
1	1.7758	1
2	0.94625	1.87667107
4	0.54996	3.2289621063
8	0.40544	4.3799329124

Dataset 3
# of iterations = 38
chunksize: 10000

Thread	Time	Speedup
1	4.93079	1
2	2.5236	1.9538714535
4	1.39284	3.5400979294
8	1.13423	4.3472576109

2. *Experiment with the scheduling options (dynamic and static), choosing a chunk size to get fast run times.*

   All on 8 threads

   Dataset 1

	Static		**Dynamic**	
Chunksize	Time	Speedup	Time	Speedup
1	0.00559	4.5706618962	0.00731	3.4952120383
10	0.00455	5.6153846154	0.00431	5.9280742459
100	0.00455	**5.6153846154**	0.00415	**6.156626506**
1000	0.01001	2.5524475524	0.0097	2.6340206186
10000	0.02658	0.9612490594	0.02658	0.9612490594
100000	0.02652	0.9634238311	0.02645	0.965973535

Dataset 2

	Static		**Dynamic**	
Chunksize	Time	Speedup	Time	
1	1.04673	1.6924135164	1.55299	1.1407027734
10	0.67653	2.618509157	0.68959	2.5689177627
100	0.4532	3.908870256	0.44105	4.0165514114
1000	0.403	**4.3957816377**	0.40368	**4.3883769322**
10000	0.44204	4.0075558773	0.44473	3.9833157197
100000	0.75676	2.3409006819	0.78691	2.2512104307

Dataset 3

	Static		**Dynamic**	
Chunksize	Time	Speedup	Time	Speedup
1	2.69868	1.8246179614	3.15172	1.5623405632
10	1.632	3.0171936275	1.56002	3.1564082512
100	1.26771	3.8842164217	1.20288	4.0935587922
1000	1.19526	4.1196559744	1.13779	4.32774062
10000	1.17125	**4.2041067236**	1.13275	**4.3469962481**
100000	1.68652	2.9196570453	1.66451	2.9582639936

3 Pagerank with dangling nodes

1. *Write down the corresponding algorithm.*

Function Pagerank_with_dangling

```
Input: n, rbegin, colindex, value, threshold, a
Output: x
Global variables: total, max_error, gamma, y[]
```

```
while (max_error > threshold) {   //termination condition
                max_error = 0;
                total = 0;

#pragma omp parallel shared(n,a) reduction(+:total)
{

                int k;                           //counteer
                int k1,k2,j;          //index
                double tmp;         // updated x[i]

#pragma omp for
          for (i = 0; i < n; i++){
                          tmp = 0;
                          k1 = rbegin[i];
                          k2 = rbegin[i+1]-1;
                          if (k2 < k1){
                                  continue;
                          }
                          for (k = k1; k <= k2; k++){
                                  j = colind[k];
                                  tmp += value[k]*x[j];
                          }
                          y[i] = a*tmp;
                          total += a*tmp;
                }
}// end parallel

        gamma = 1 - total;

#pragma omp parallel shared(n,gamma) reduction(max:maxerr)
{
        double error, tmp, prev;

#pragma omp for
                for (i = 0; i < n; i++){
                          tmp = y[i];
                          prev = x[i];
                          tmp = tmp + gamma/n;
                          error = fabs(tmp - prev);
                          if(error > max_error){
                                  max_error = error;
                          }
                          x[i] = tmp;
                }// end for loop
```

```
}// end parallel

}// end while loop

        return x;
```

2. *Time Complexity*

   There are two parallel region. One is for sparse matrix multiplication; another is for updating the pagerank vector. For the first one, the time complexity for each thread is $\lceil n/p \rceil (2\Delta + 4)$. For the second parallel region, the time complexity for each thread is $2 \lceil n/p \rceil$. Therefore, for one iteration, the time complexity is $\lceil n/p \rceil (2\Delta + 6)$. Hence, when dynamic scheduling is used (only for the first region), the time complexity becomes $2 \lceil nnz(A)/p \rceil + 6 \lceil n/p \rceil$.

Performance (default scheduling)

All on 1 thread

	Time
Data1	0.02616
Data2	1.96057
Data3	5.13793

All on 8 thread

	Time	Speedup
Data1	0.00652	4.0122699387
Data2	0.47	4.1714255319
Data3	1.28149	4.0093406894