Ankit Goyal
ankitgoyal@utexas.edu

Homework 1
Software Multicore Processors

In this report various variants of page rank algorithm has been implemented and analyzed for performance. All the programs were ran on the same hardware with 16 cores. Code was compiled using icpc compiler.
The two variants that has been implemented are:
a) Two copy page rank: where two copies of the page rank vector are kept, R_i and R_{i+1}. In this algorithm, the value of next page rank vector depends on the page rank vector at the previous step.
b) One copy page rank: where only one copy is maintained and at each step vector is updated on its place.

Data Structures Used

Given a graph G, all the nodes have been represented as an index in a vector. Each node has two vectors associated with it. One vector contains out degree count of a node and the other contains the in-vertices of a node.

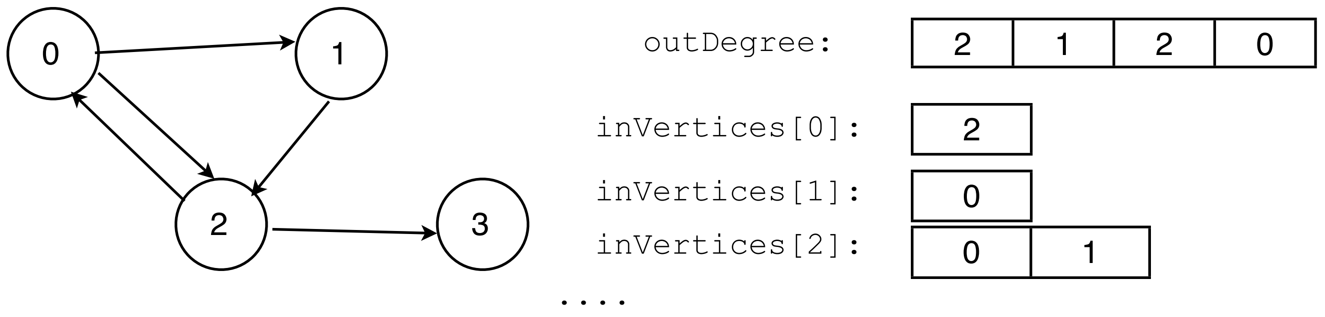


Figure 1.

Two Copy Algorithm

Two more vectors are maintained for page rank at previous step and the next step: `page_rank_previous and page_rank_next`

Page rank is calculated using:

$$R_{i+1}[v] = \frac{1-\beta}{|V|} + \beta * \sum_{u \in in\_neigh(v)} \frac{R_i[u]}{out\_deg(u)}$$

Ankit Goyal
ankitgoyal@utexas.edu

So to calculate rank for a node we need all the vertices that are incoming to a node and the out degree of a node. Having separate vector for these two makes getting a node's out-degree a constant order operation.

To calculate a rank, iterate over all nodes and keep adding contribution from their neighbors from the previous step page rank in case to Two Copy page rank algorithm.

```
  while(tolerence > .0001 and iterations < 100){
  tolerence = 0.0;
  int i,j, tid;
#pragma omp parallel for private(i,j,tid) shared(page_rank_next, page_rank_previous,
Vcount, constant_part, damping_factor, inVertices, outDegree)
    for(i=0;i<Vcount;i++){
      double temp = 0.0;
      for(j=0; j < inVertices[i].size(); j++){
        temp += page_rank_previous[inVertices[i][j]] / outDegree[inVertices[i][j]];
      }
      page_rank_next[i] = constant_part + (damping_factor *  temp);
    }

#pragma omp parallel for private(i) shared(page_rank_next, page_rank_previous, Vcount)
reduction(+: tolerence)
    for(i =0; i < Vcount; i++){
      tolerence += abs(page_rank_next[i] - page_rank_previous[i]);
      page_rank_previous[i] = page_rank_next[i];
    }
```

Snippet. 1

As shown in the Snippet 1. The inner loop doesn't have any data dependencies. Each thread is reading from read-only page_rank_previous and writing to different index at page_rank_next. Implicit synchronization is used.

Results:

The algorithm was ran for different number of nodes. The parallel version was ran for various number of threads. Table. 1 shows the runtime of algorithm for sequential algorithm(2nd column) and parallel algorithm for different number of nodes on different number of threads.

213 108 76

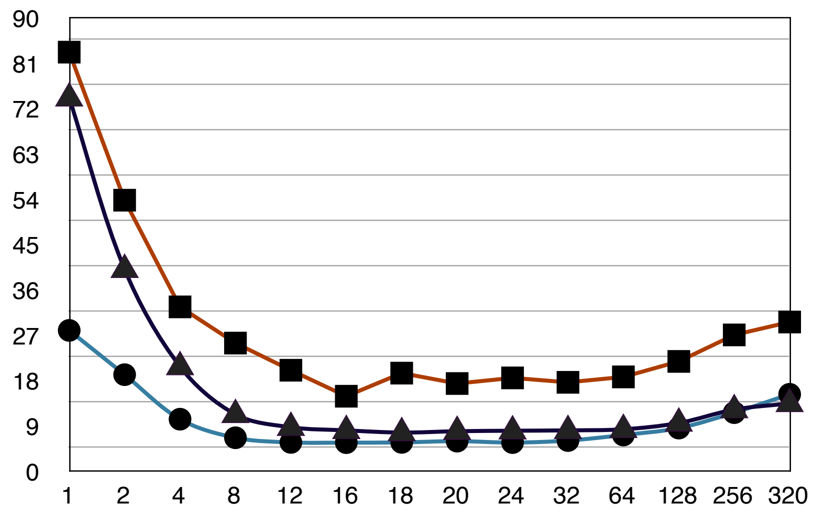
Ankit Goyal
ankitgoyal@utexas.edu

n	***Seq.***	*1*	2	4	8	12	16	18	20	24	32	64	128	256	320
T_n *(nodes = 33554432)*	75	74.1	40	20.7	11.2	8.69	8.07	7.63	7.9	7.99	8.07	8.28	9.53	12.2	13.4
T_n *(nodes = 23947347)*	28.1	27.9	19.1	10.3	6.6	5.68	5.63	5.68	5.93	5.64	6.05	7.15	8.52	11.6	15.2
T_n *(nodes = 16777210)*	82.9	83.2	53.8	32.6	25.4	20	14.8	19.4	17.4	18.5	17.6	18.7	21.8	27	29.6

Table. 1

Observations:

- It can be seen that run time doesn't depend only on the number of nodes but it also depends on the type of graph(sparsity, clusters, etc).
- The run times for sequential algorithm(2nd column) are almost equal to the run time of parallel algorithm running on single thread as expected.
- In almost all cases, the optimal rate is achieved when the number of threads is equal to the number of chores(16 in our case). Can be confirmed from the plot.
- In plot 1 it can be seen that as the number of threads increases the time taken to converge decreases to a limit and then it becomes almost constant or even starts to increase due to the overhead of threads and context switching.



Plot 1. (run time vs Number of Threads)

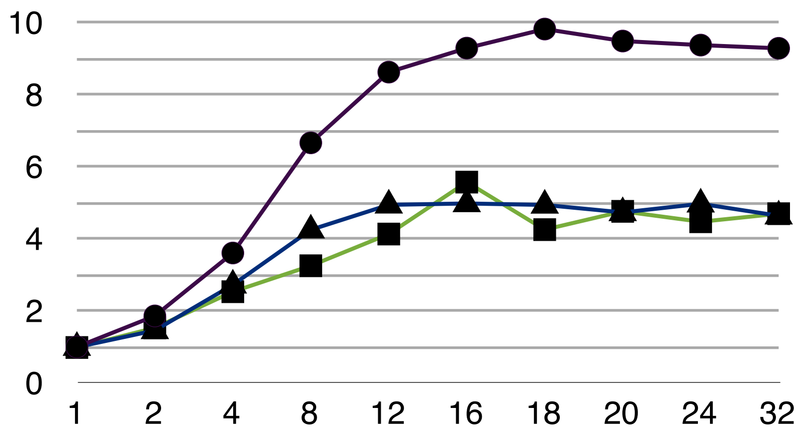
Plot shows runtime of a parallelized page rank algorithm for different inputs vs. different number of threads on a 16 core machine.

● 23947347 nodes
▲ 33554432 nodes
■ 16777210 nodes

Note that the after a certain number of threads runtime starts to decrease due to increase in overhead due to threads.

Ankit Goyal
ankitgoyal@utexas.edu

Speedup and Scalability Plots



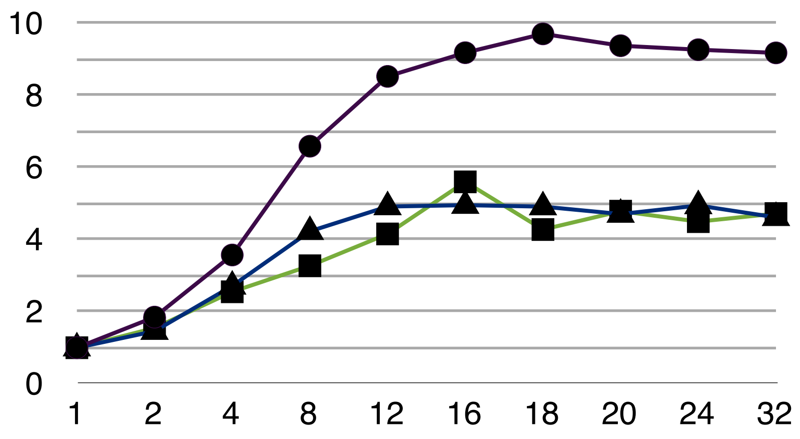
Plot. 2 (Speedup)

It can be seen that the maximum speedup for each input is bounded and after a certain point it becomes constant.

Speedup is higher for larger number of nodes.

$$Speedup = \frac{T_s}{T_n}$$

● 23947347 nodes ▲ 33554432 nodes ■ 16777210 nodes



Scalability is higher for larger number of nodes.
Scalability is also bounded and starts to decrease after a point.

Maximum at threads equal to number of cores.

$$Scalability = \frac{T_1}{T_n}$$

Plot. 3 (Scalability)

One Copy Algorithm

For this version of the page rank algorithm only one copy of page rank is maintained so instead of `page_rank_previous` and `page_rank_next vectors` only `page_rank vector` is used.

Parallelizing this algorithm introduces a data dependency since nodes are getting updated in different threads and we have a restriction to maintain only one copy of the `page_rank` vector.

Ankit Goyal
ankitgoyal@utexas.edu

```
  double page_rank_previous_i=0.0;
  while(tolerence > .0001 and iterations < 100){
  tolerence = 0.0;
  omp_set_num_threads(atoi(argv[2]));
  int i,j, tid;
#pragma omp parallel for private(i,j,tid, page_rank_previous_i) shared(page_rank,
Vcount, constant_part, damping_factor, inVertices, outDegree) reduction(+: tolerence)
    for(i=0;i<Vcount;i++){
      tid = omp_get_thread_num();
      page_rank_previous_i = page_rank[i];
      double temp = 0.0;
      for(j=0; j < inVertices[i].size(); j++){
        temp += page_rank[inVertices[i][j]] / outDegree[inVertices[i][j]];
      }
      page_rank[i] = constant_part + (damping_factor *  temp);
     tolerence += abs(page_rank[i] - page_rank_previous_i);
 }
    iterations++;
  }
```

However we can ignore this data dependency since our algorithm is random enough and doesn't effect the output.

Proof: Let's assume we are calculation page rank for node *A* and we are page rank values of incoming node *B***.** Now some other thread may or may not be updating the value of B. So either the old value of B will be used or new value of B will be used and it doesn't matter given our algorithm.
Page rank obtained by ignoring the data dependency is very close to the page rank obtained in the parallel two copy algorithm, shown using Kendal Tau Distance.

n	***Seq.***	1	2	4	8	12	16	18	20	24	32	64	128	256	320
T_n (nodes = 33554432)	43.7	43.7	24	12.6	6.69	4.98	4.52	4.51	4.6	4.73	4.73	4.81	5.33	5.91	6.5
T_n (nodes = 23947347)	13.1	13	8.93	4.76	2.78	2.3	2.26	2.14	2.52	2.58	2.58	2.86	3.76	4.36	6.61
T_n (nodes = 16777210)	84.6	84.2	47.5	30.3	18	14.7	10.3	12.3	10.4	11	11.5	11.7	12.5	15.4	15.9

Table. 2

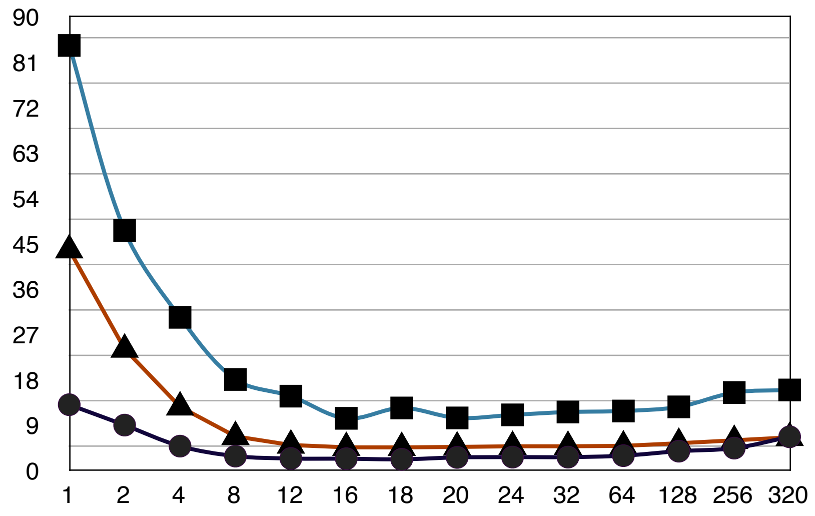
Table 2. shows the runtime of one copy page rank algorithm for the same set of inputs as two copy page rank algorithm. Note that the runtime for one copy is smaller than the run time of two copy, since only one copy is maintained at each step and overhead of maintaining two vectors is not encountered.

Ankit Goyal
ankitgoyal@utexas.edu

Observations:

•Significant improvement in performance can be observed in Plot. 4 in parallel version.

•Again note that the optimal runtime is achieved when the number of threads is equal to the number of cores (16, in our case)

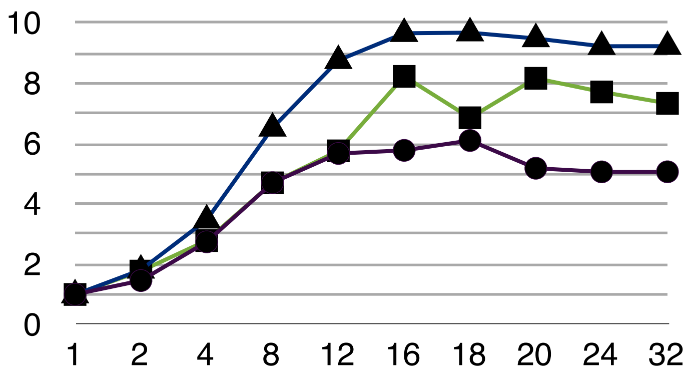


• Note that the performance difference is different for different number of nodes and are based on the graph type.

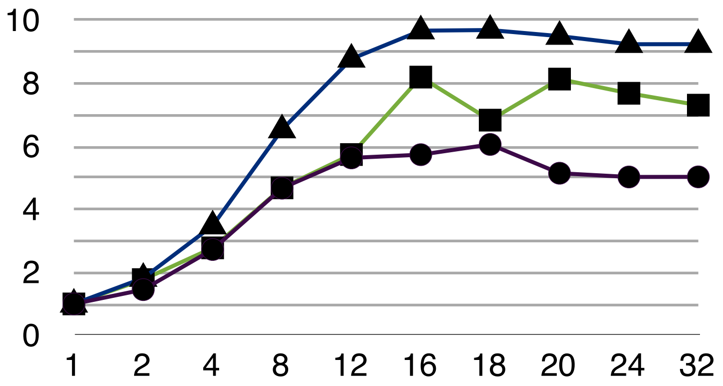
•Again the runtime of sequential program is equivalent to the parallel program with one thread.

•Again note that the maximum speedup (Plot 5) is bounded irrespective of the algorithm as suggested by Amdahl's Law.

Plot. 4 (runtime vs number of threads)



Plot. 5 (Speedup)



Plot. 6 (Scalability)

Results:

•It can be seen that runtimes are better than the parallel version of the algorithm, since the overhead of maintaining two copies of shared variable page rank is not incurred.

Ankit Goyal
ankitgoyal@utexas.edu

• Table. 3 shows the average degree of each graph. Note that the graph with higher average degree has a higher improvement in the runtime compared to the other graphs.

	Nodes	Edges	Average Degree ($\frac{E}{V}$)
■	16777202	134217728	8
▲	33554430	134217728	4
●	23947344	57708624	2

Table. 3 (Average Degree)

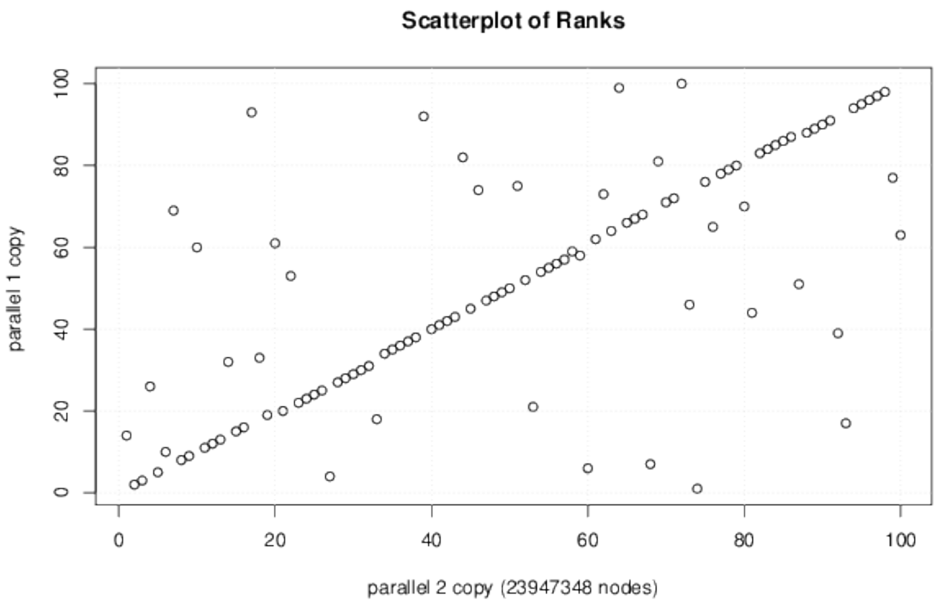
• In case of higher degree each thread does more work and the overhead of the threads is amortized by the gain due to parallelization.

• Note that data dependency was assumed to not effect the final page rank values during the parallelization of the one-copy version and the results and the effectiveness of the assumption can be verified using Kendal Tau Distance.

Table. 4 shows the Kendal Tau Distance values between different versions of the page rank algorithm on a particular random run.

Algorithms Compared	Kendall Tau Distance(first 100 nodes) (total -16777202 nodes)	Kendall Tau Distance(first 100 nodes) (total -23947348 nodes)
Sequential (two copy) vs Sequential (one copy)	2	15
Parallel (two copy) vs Parallel (one copy)	3	23
Sequential (one copy) vs Parallel (one copy)	5	19
Sequential (two copy) vs Parallel (two copy)	0	0

Table. 4 (Kendall Tau Distance)



On the side is the scatterplot of ranks to show that the page rank difference is usually due to adjacent nodes being off by 1 or 2 distance.

There are 19 nodes with page rank off by at most 4. Technically only 1 node can be off by 4; since the sum is 23. Hence the page rank values are consistent enough.

Scatter Plot made using: http://www.wessa.net/rwasp_kendall.wasp