1.)

Proc Global_Space_Parallel_Page_Rank:

Inputs:

value[]: link probability parse matrix using the compressed row data structure
rowbegin[]: where each row begins in the value array
colindex[]: index of the column

```
1       for row i=0 to n-1:
2               x[i] = 1/n;
3       end for
4       gamma = ((1-ALPHA)/n)
5       while (y – x > 10^-5)
6               for row i=0 to n-1 do  in parallel:
7                       y[i] = 0
8                       k1 = rowbegin[i];
9                       k2 = rowbegin[i+1] -1
10                      if k2 > k1:
11                              for k = k1 to k2 :
12                                      j = colindex[k]
13                                      y[i] += ALPHA * value[k] * x[j]
14                              end for
15                      end if
16                      y[i] += gamma
17              end for
18      end while
end proc
```

Static Scheduling Time Complexity:

When using a static schedule the computations in the algorithm are divided amongst threads by rows. Each thread receives the same number of rows and does their work individually. The analysis of the algorithm's time complexity for one iteration can be seen below:

$$3 \text{ (number of computations on line \#13)} * (n/P) * \Delta + (n/p)(\text{operation on line 16}) = (n * \Delta / p)$$

However, this is not always the best method for scheduling. As rows have varying amount of zeros, some threads will have to do significant more computations than others. To combat this problem dynamic scheduling can be used.

Dynamic Scheduling Time Complexity:
When using a dynamic schedule the computations in the algorithm are divided evenly amongst the threads. Therefore, each thread is required to perform the same amount of work. The analysis of the dynamically scheduled algorithm's time complexity for one iteration can be seen below:

3 (number of computations on line 13) * (nnz(A)/P) + (n/p)(operation on 16)
=
(nnz(a)/P) + (n/p)

2.)

Size	#Non-zeros	Iterations	Threads	Time	Speedup
4039	176468	12	1	.03029	N/A
4039	176468	12	2	.01856	1.63200431
4039	176468	12	4	.01412	2.145184136
4039	176468	12	8	.01396	2.169770774
325729	2207671	42	1	1.84144	N/A
325729	2207671	42	2	1.04256	1.766267649
325729	2207671	42	4	.67007	2.748130792
325729	2207671	42	8	.53054	3.470878727
489800	9131686	38	1	6.01290	N/A
489800	9131686	38	2	2.99691	2.006366558
489800	9131686	38	4	1.72492	3.485900795
489800	9131686	38	8	1.42826	4.209947769

3.)

Proc Global_Space_Parallel_Page_Rank_Dangling_Nodes:
Inputs:

value[]: link probability parse matrix using the compressed row data structure
rowbegin[]: where each row begins in the value array
colindex[]: index of the column

```
1     for row i=0 to n-1:
2         x[i] = 1/n;
3     end for
4     gamma = 0
5     while (y – x > 10^-5)
6         for row i=0 to n-1 do  in parallel reduction(+:gamma):
7             y[i] = 0
8             k1 = rowbegin[i];
9             k2 = rowbegin[i+1] -1
10            if k2 > k1:
11                for k = k1 to k2 :
12                    j = colindex[k]
13                    y[i] += ALPHA * value[k] * x[j]
14                end for
15            end if
16            gamma = y[i]
17        end for
18        gamma = (1-gamma)/n
19        for i=0 to n-1 do in parallel:
20            y[i] += gamma
21        end for
22    end while
end proc
```

Dynamic Scheduling Time Complexity:
The time complexity for dealing with dangling nodes will be very similar to the original algorithm. However, gamma must be calculated based on the value in the y array and then added individually to each value in the y array. To calculate the value of gamma the algorithm uses reduction. This way at the end of the for loop that calculates y, gamma will contain a summation of all values of y. Then after exiting the for loop the summation is subtracted from one and divided by n to get the correct gamma value. Finally, y is iterated through and every value is adjusted by adding gamma to it. The time complexity of this algorithm can be seen below:

3 (number of operations on line #13) * (nnz(A)/P) + n(reduction sum of gamma value) + 2(operations on line #18) + (n/p) (for loop on line #19)

=

(nnz(A)/P) + n + (n/p)