Vamos a considerar una matriz de dimensiones 100×100 y realizaremos 3000000 iteraciones del algoritmo (queremos que la matriz no sea tan grande ya que con el algoritmo entregado para la lectura se tarda demasiado puesto que debe leerla toda), puesto que con estos parámetros y con 1 procesador se demora 164 segundos, lo cual es práctico para los experimentos. Los resultados para calcular el strong scaling se muestran en la tabla siguiente

Tamaño de Matriz	Iteraciones	Procesadores	Tiempo	Aceleración	Eficiencia
100×100	3000000	1	164s	1.00	1.00
100×100	3000000	2	98s	1.71	0.86
100×100	3000000	3	82s	2.04	0.68
100×100	3000000	4	71s	2.36	0.59
100×100	3000000	5	67s	2.50	0.50
100×100	3000000	6	67s	2.50	0.42
100×100	3000000	7	67s	2.50	0.36
100×100	3000000	8	72s	2.27	0.28
100×100	3000000	9	65s	2.52	0.28
100×100	3000000	10	66s	2.48	0.25
100×100	3000000	11	63s	2.6	0.24
100×100	3000000	12	68s	2.41	0.20
100×100	3000000	14	72s	2.27	0.16
100×100	3000000	16	73s	2.24	0.14
100×100	3000000	18	84s	1.95	0.11
100×100	3000000	20	87s	1.88	0.09

Y no logré conseguir más de 20 procesos en el cluster así que la información quedará hasta aquí. Al observar la aceleración y la eficiencia podemos concluír que el trabajo en paralelo fue beneficioso para el tiempo de ejecución del algoritmo. Sin embargo, a partir de los 5 procesadores dejó de ser significativa la aceleración e incluso bajó.

Lo anterior se puede deber a muchos factores, el principal corresponde al tiempo en que se demoran en comunicarse los distintos procesos. De hecho, si pedimos tantos procesos que SLURM debe asignar dos nodos distintos, entonces la comunicación entre procesos tendrá que pasar por la red del clúster en lugar de solo la memoria compartida del nodo, la cual tiene mayor latencia de comunicación y por tanto empeora la eficiencia (esto es lo que debería estar pasando al usar 18 procesadores por ejemplo).

Lo mejor en este caso sería probar con otros parámetros tales que disminuyan la comunicación, podríamos por ejemplo disminuir las iteraciones y aumentar el tamaño de la matriz, por lo que es lo que haremos para los nuevos experimentos. El único detalle es que el formato que se nos entregó para trabajar el input implica tener que leer todos los datos (aún cuando no hayan que almacenarlos), por lo que te todas formas podría ser complicado tener que trabajar con matrices demasiado grandes. Realicemos el experimento

Tamaño de Matriz	Iteraciones	Procesadores	Tiempo	Aceleración	Eficiencia
10000×10000	300	1	211s	1.00	1.00
10000×10000	300	2	137s	1.54	0.77
10000×10000	300	3	108s	1.95	0.65
10000×10000	300	4	95s	2.22	0.55
10000×10000	300	5	88s	2.40	0.48
10000×10000	300	6	82s	2.57	0.43
10000×10000	300	7	79s	2.67	0.38
10000×10000	300	8	75s	2.81	0.35
10000×10000	300	9	74s	2.85	0.32
10000×10000	300	10	72s	2.93	0.29
10000×10000	300	11	70s	3.01	0.27
10000×10000	300	12	69s	3.06	0.26
10000×10000	300	14	67s	3.15	0.23
10000×10000	300	16	65s	3.25	0.20
10000×10000	300	18	65s	3.25	0.18
10000×10000	300	20	64s	3.30	0.17

Podemos notar que cuando la cantidad de procesadores es chica tiene una peor eficiencia que en el experimento anterior. Sin embargo, cuando aumentamos el número de procesadores no se estanca tan bruscamente y por tanto logra mejores eficiencias que en el experimento anterior. Esto tiene mucho sentido dado lo discutido anteriormente.

Por último, realicemos un experimento para comprobar la escalabilidad débil, para esto realizaremos una cantidad fija de iteraciones (por ejemplo 300000) y aumentaremos el tamaño de la matriz en $\sqrt{p}$ donde p son los procesadores, puesto que nuestro algoritmo es cuadrático en función del tamaño de la matriz

Tamaño de Matriz	Iteraciones	Procesadores	Tiempo	Aceleración	Eficiencia
100×100	300000	1	18s	1.00	1.00
141×141	300000	2	20s	1.8	0.9
173×173	300000	3	21s	2.57	0.85
200×200	300000	4	22s	3.27	0.81
223×223	300000	5	22s	4.05	0.81
244×244	300000	6	24s	4.5	0.75
264×264	300000	7	23s	5.46	0.78
282×282	300000	8	24s	6	0.75
300×300	300000	9	27s	5.94	0.66
316×316	300000	10	25s	7.2	0.72
331×331	300000	11	25s	7.92	0.72
346×346	300000	12	26s	8.28	0.69
374×374	300000	14	28s	8.96	0.64
400×400	300000	16	27s	10.56	0.66
424×424	300000	18	29s	11.16	0.62
447×447	300000	20	30s	12	0.6

En este experimento se ve que la eficiencia estuvo mucho más cerca de la eficiencia teórica optimista, por lo que podemos concluír que nuestro algoritmo (y en particular, nuestra implementación) tiene una mejor escalabilidad débil que una escalabilidad fuerte.