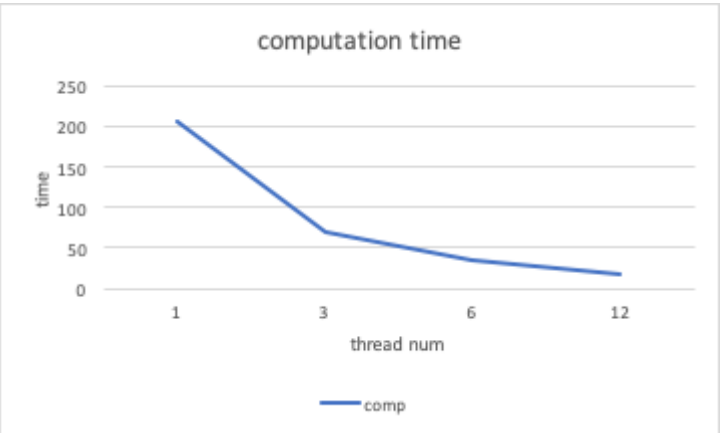
PPHW3

implementation

1. Algo: Floyd-Warshall
2. Implementation:
   - 利用adj. matrix當作我的data struct，並且在計算i to j的時候，利用omp平行計算點i到所有的其他點j的shortest path
   - 如果k到j的距離是INF，則skip
   - 使用static scheduler
   - 將INF定義為1073741823
   - io得部分即使使用omp的order仍然無法平行讀取，因此io部分不做平行
   - array在一開始就會進行初始化的動作，所有的點利用uninitialized_fill_n()進行matrix的初始化，再將對角線設定為0
   - adj. matrix是使用1D的array模擬2D array，如此一來只需access一次memory
3. Time complexity: 沒有平行化的scheduler是O(N^3)，平行化以後的time complexity是O(N^2 * (N/P))
4. Design testCase:
   想要創建一個sparse matrix，使用6000個vertices與6000個edge來創建一個graph，其中src和dst跟weight都是透過random來選擇

Experiment & Analysis

1. System Spec: apollo
2. Strong scalability:

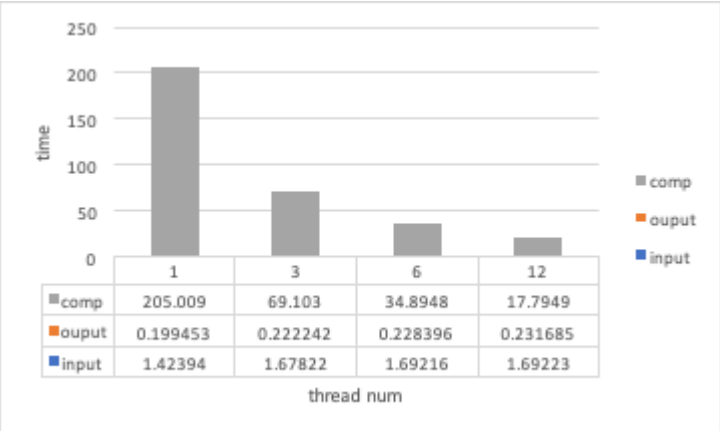


從圖表來看使用幾條thread就有達到幾倍的加速，且每個thread的work loading幾乎相同

max	min	diff
193.869	193.869	0
65.0848	65.0543	0.0305
33.0342	33.0243	0.0099
19.6208	19.5514	0.0694

所以工作量很平衡，達到非常好的加速效果

3. Time profile:



	1	3	6	12
comp	205.009	69.103	34.8948	17.7949
ouput	0.199453	0.222242	0.228396	0.231685
input	1.42394	1.67822	1.69216	1.69223

從圖表看出，bottleneck在於計算，只要能夠加速計算，就能夠有效的減少程式的運行時間
從這邊也看出因為我們讀的限制所以只能只用for loop去讀取src與dst，但是我們寫入資料的時候可以根據adj. matrix的內容直接寫入disk，因此效能上寫入的效能會比讀取來的好很多

Experience & conclusion

1. 使用Floyd-Warshall algo.雖然能達到很好的scalabilty，但是他有個缺點就是相較於使用Dijkstra algo計算量比較大，而在我們的input當中，因為所有的點之間的距離都是正整數，所以使用Dijkstra會比Folyd-Warshall來得好
2. 在思考Dijkstra的時候，會發現一個問題是，雖然理論上Dijkstra的速度使用heap的data structure會比較快，但是更新距離的那一步，需要O(n)的時間來更新所有的node到source的距離，且共要更新n次，所以時間仍可能達到O(n^2)，相較於Folyd-Warshall的優勢反而在pop完以後需要更新的點可以逐漸減少，而Folyd沒有此優勢
3. 嘗試在進行for loop的時候同時平行i與j(透過collapse(2))，但是time反而上升很多，但是原因未知