Jacobi Method Using OpenMP

Abstract

이번 과제에서는 Jacobi method의 수렴 조건을 만족하는 랜덤 행렬을 만들고, Jacobi method를 사용해 해를 구한 다음 답과 일치 여부를 확인하였다. 이후, 행렬 크기를 2배씩 늘리며 스레드 개수에 따라 시간이 얼마나 소요되는지 확인하였다. 이때, 랜덤 행렬에 대해 계산하므로 편차가 있을 수 있어 100번 반복 후 평균값을 사용하였다. 이후 병렬화가 유의미하게 계산 시간을 줄여주는 크기에서 schedule 방법에 따른 계산 시간을 비교하였다.

야코비 방법(Jacobi method) 은 연립 일차 방정식의 해를 구하는 방법으로 반복적으로 해를 계산하는 방법이다. 연립 일차 방정식을 $\mathrm{Ax} = \mathrm{b}$ 라 할 때, 수렴하는 충분 조건으로 식(1) 이 성립해야 하며, (k+1) 번째 x값은 식(2)와 같은 방식으로 찾는다.

$$|\mathrm{A_{ii}}| > \sum_{j!=i} |A_{ij}| \tag{1}$$

$$\mathrm{x_i^{k+1}} = \frac{1}{A_{ii}}(b_i - \sum_{j!=i} A_{ij}x_j^k) \tag{2}$$

OpenMP는 공유 메모리 다중 처리 API(Application programming interface) 로 각 스레드들이 전역 메모리 공간을 공유할 때 병렬 프로그래밍을 하는 API이다. 이번 과제에서는 windows 환경에서 c++ 언어를 사용해 구현하였다.

랜덤행렬을 생성할 때, 행렬 값이 [-1,1] 사이에 포함되게 하였으며, A, x 를 생성한 후 b를 계산해 저장하였다. 매번 다른 행렬을 생성하게 하기 위해 randomseed 를 ctime의 time() 함수를 통해 대각 성분에 대각에 있지 않은 다른 성분들의 합+1을 더해주어 절대값이 합보다 크도록 만들었다. 다만, 병렬화 speedup 을 확인할 때 중요한 것은 Jacobi method 부분이지 행렬 생성 부분은 아니라 판단하여 schedule(auto) 로 설정한 후 바꾸지 않았다.

Jacobi method 를 사용해 해를 구할 때, iteration 은 순차적으로 이루어져야 하므로 병렬화를 하지 않았다. 여러 부분을 병렬화 할 수 있지만, fork & join 에 시간이 소요되므로 병렬화하여 각 스레드별 동작이 최대로 이루어질 수 있도록 병렬화 하기 위해 식(2)의 i에 대해 병렬화를 진행하였다. K 번째와 K+1 번째 값 사이 MSE값을 구한 뒤, 그 값이 tolerance 로 설정한 값보다 작을 경우 Jacobi method 로 동작한 시간을 출력하게 하였다. 시간은 omp_get_wtime() 함수를 사용하였다. Jacobi method 로 구한 값이 실제값과 같은지는 get_error 함수를 만들어 비교하였으며, reduction 지시어를 사용해 계산을 병렬화 하였다. 이후, 행렬 차원과 스레드 수에 따른 속도를 비교할 때에는 schedule(static, N_chunks=16)을 사용해 비교하였으며, 행렬 차원이 4096, 즉, 4096 * 4096 행렬에 대해 schedule설정에 따른 시간을 비교하였다.

Figure1에서 확인할 수 있듯이, 행렬 차원이 1024 이상이 되어야 병렬화로 계산하는 것이 더 빨랐다. 행렬 차원이 작은 경우 fork, join을 하는데 overhead 가 크기 때문이라고 생각한다. Figure2에서 Speedup =(병렬화 후 실행시간)/(스레드 1개 실행시간)을 계산했을 때, 행렬 차원 4096 이상에서 개선부분 비율이 0.983 이라 가정했을 때 피어슨 상관계수 값이 0.943로 Amdahl's law 가 잘 성립하였다. 스레드 32개를 사용했을 때 성능이 Amdal's law 를 따르는 값보다 낮은 것은 컴퓨터에서 다른 프로그램이 활성화된 상태이기에 CPU의 스레드 최대 개수 32개를 사용할 때 overhead 가 있었던 것으로 보인다.

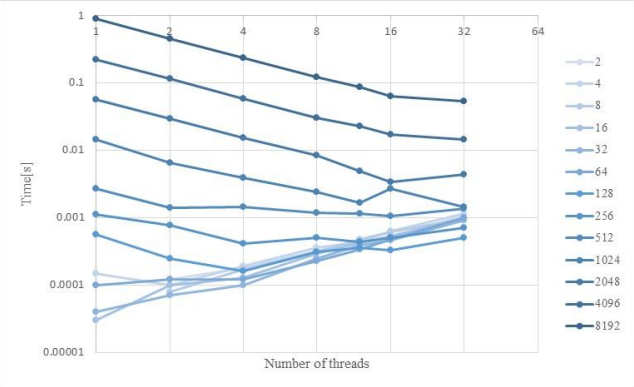

Figure 1

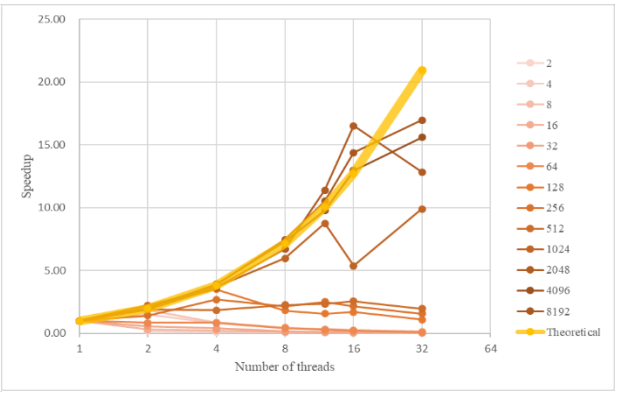

Figure 2

Schedule 설정은 static, dynamic, guided, auto로 설정하여 비교하였다. Static, dynamic, guided의 경우 N_chunks 를 2의 배수로 하여 L2 cache 를 이용하려 하였다. 차원이 4096인 행렬에 대해 스레드 수를 32로 고정하였으며 100번 실험한 뒤 평균과 표준편차를 구하였다. 오차막대의 크기는 표준편차를 사용하였다. Figure3 에서 알 수 있듯이 평균값은 dynamic 값이 작았으나 표준편차는 가장 컸다. 다만 평균에서 표준편차를 더한 값이 static과 dynamic에서 비슷한 것으로 보아 dynamic 방법이 작업이 끝난 스레드에 일을 배정해주어 speedup 에 이점이 있는 것을 확인할 수 있었다. N_chunk 값에 따른 차이는 거의 없는 것으로 보였다. Auto 혹은 guided 로 설정한 경우 평균값이 다른 방법에 비해 작지 않았지만 표준편차가 매우 작았다. 편차가 거의 없이 빠르게 동작하는 상황이라 생각한다.

따라서, 최종적으로 if clause 를 사용하여 행렬 차원이 1024일 때 병렬로 동작하게 하였으며 schedule 은 guided 를 사용해 계산시간을 빠르게 하였다.

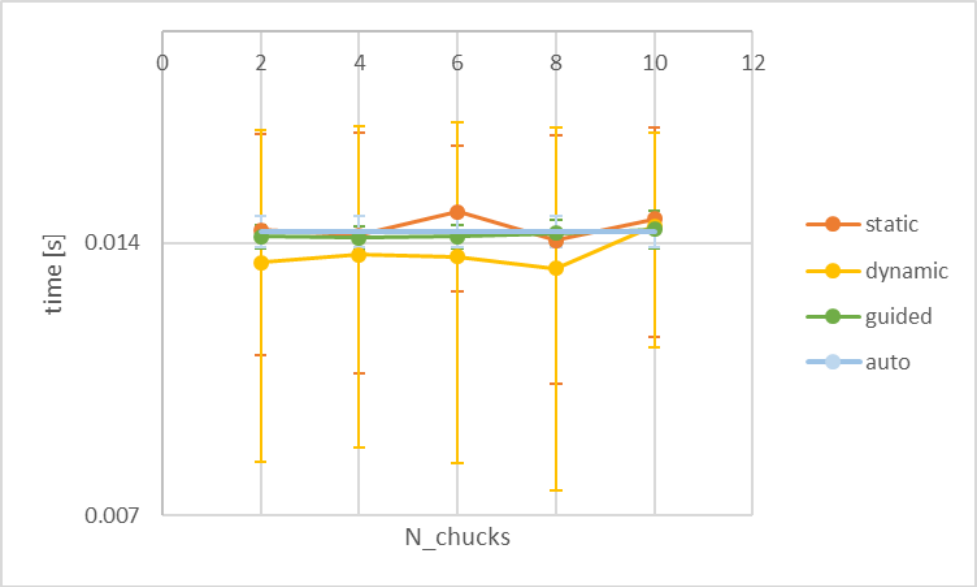
0
2
4
6
8
10
12
time [s]
0.014
0.007
N_chucks
static
dynamic
guided
auto

Figure 3