Lab03 Report

201920739 소프트웨어학과 박지윤

1) Describe how to parallelize your game of life

Game of life를 구현하기 위해 다음과 같은 방법을 택하였다. 먼저, thread의 개수를 입력을 받으면, game of life matrix의 계산을 적절하게 thread별로 분배를 해주어야 한다. 이때, game of life matrix의 row 개수를 thread의 개수로 나눈 몫과 나머지를 구하고, 각각의 thread에 몫만큼의 row를 배분해준 후, 첫 번째 thread부터 차례대로 1씩 추가로 row를 배분해주어, 나머지 개수만큼의 thread가 추가로 row를 한 개씩 받도록 하였다. 즉, 20개의 row가 있다고 할 때, 만약 5개의 thread로 돌린다면 각 thread 당 4개의 row를 배분받게 된다. 만약 6개의 thread로 돌린다면 차례대로 4개, 4개, 3개, 3개, 3개, 3개를 배분받게 된다.

처음에는 matrix의 원소별로 균등하게 배분하는 방식을 생각하였으나, 경우의 수가 너무 많고, 오히려 균등하게 배분하기 위해 계산하는 과정에서 시간이 더 소모될 것 같아 thread당 처리해야 할 범위를 row를 기준으로 주었다.

Multi-threading을 할 때, m_Temp matrix에 game of life의 규칙에 따른 각 원소의 live/dead 여부를 기록한 것을 m_Grid로 update해야 한다. 이때, m_Temp에 각 원소의 live/dead 여부를 판별하여 기록하는 작업을 multi-threading으로 처리했기 때문에 thread간 처리 속도가 다를 경우, 원치 않는 값이 m_Grid로 update가 될 수 있다. 따라서 각각의 thread가 m_Temp에 각 원소의 live/dead 여부를 기록한 후에 m_Temp의 값이 m_Grid로 update가 될 수 있도록 pthread_barrier_wait()를 걸어주었다.

내가 구현한 코드에서는 m_Temp의 값을 m_Grid로 update하는 작업도 각각의 thread가 배분받은 row의 범위만큼 update를 담당하게 된다. 이 경우, 위와 같은 이유로 m_Grid가 완전히 update되기 전에 어떤 thread가 m_Grid의 값을 보고 m_Temp에 각 원소별 live/dead 여부를 기록하면 안 된다. 따라서 m_Temp의 값을 m_Grid로 update하는 작업을 모든 thread가 마쳐야 다음 loop를 수행할 수 있도록 이 부분에도 pthread_barrier_wait()를 걸어주었다.

Thread가 해야할 일을 다 처리한 경우, thread를 회수해주어야 컴퓨터 리소스를 효과적으로 관리할 수 있다. 따라서 모든 작업이 끝난 후에, pthread_exit()로 thread를 종료해주었다.

Main thread는 multi-threading 작업이 끝나기 전까지는 수행해야 할 작업이 없으므로, block상태에 있어야 한다. 따라서 main thread를 block상태에 있도록 하기 위해 pthread_join()을 사용하였다.

2) Make a performance comparison by increasing the number of threads

과제에서 작성한 코드에서는 크게 sequential, single thread, multi thread의 3가지 종류가 있다. 이를 한 번 비교해보고자 한다.

먼저, puf-qb-c3_4290_258 파일을 대상으로, 300*300 matrix를 기준으로, display는 0로 했을 때 3가지 버전을 돌렸을 때의 결과는 다음과 같다.

Generation	Sequential	Single Thread	Multi Thread (threads = 10)
0	0	0.000273	0.000759
1	0.014746	0.016695	0.002968
2	0.025248	0.027152	0.005296
5	0.055699	0.060269	0.011975

먼저, sequential과 single thread의 경우, sequential은 main thread에서 작업을 수행하지만 single thread는 main thread에서 thread를 하나 생성해서 작업을 수행하기 때문에, thread를 생성하는데 시간이 소모되어 미세하게 시간이 증가함을 확인할 수 있다. Multi thread의 경우, generation이 0인 경우에는 multi thread로 효과를 보고자 하는 계산 작업이 전혀 수행되지 않기 때문에, thread 생성하는데 시간이 소모되어 sequential이나 single thread에 비해 수행 시간이 길다. 하지만, generation이 1을 넘어가면서 계산작업이 추가되는데, multi thread인 경우 thread들끼리 나누어서 작업을 수행하기 때문에 sequential이나 single thread에 비해 수행 시간이 확연하게 짧은 것을 확인할 수 있다.

Generation	Threads=2	Threads=14	Threads=15	Threads=16
0	0.0003	0.001143	0.001183	0.001206
1	0.010003	0.002699	0.002703	0.003449
10	0.061802	0.016504	0.015624	0.019807

위 표는 생성한 thread의 수에 따른 각 generation 별 수행 시간이다. Generation이 0일 때는 앞서 서술했듯이, multi-threading의 핵심인 계산과정이 수행되지 않기 때문에, thread를 생성하는데 걸리는 시간만 측정되어 thread의 개수가 많아질수록 수행 시간이 길어진다. Generation이 1을 넘어가면서 threads의 수에 따른 수행 시간에 유의미한 결과를 얻을 수 있다. 다만, 무조건적으로 thread의 수가 많아져야 수행시간이 짧아지는 것은 아니고, thread를 생성하는데 드는 overhead보다 thread를 1개 더 생성하여 줄어드는 연산 수행 시간이 더 짧아야 수행시간이 짧아진다. 만약, thread를 1개 더 생성하여 줄어드는 연산 수행 시간보다 thread를 1개 더 생성하는데 드는 overhead가 더 커진다면 연산 속도는 증가하게 된다. 위 표에서는 thread의 개수가 15를 넘어가면서 연산 수행 속도가 증가하는 것을 확인할 수 있었다.

다음은 연산량에 따른 multithread별 수행 속도를 비교한 것이다. 이때, generation은 5로 고정하였다.

threads	300*300	500*500	1000*1000	5000*5000
10	0.011967	0.022314	0.062675	1.21275
100	0.013735	0.018199	0.044886	0.840893

위의 표에서 확인할 수 있듯이, 매트릭스의 크기가 커지면 연산량이 증가하기 때문에 수행 시간도 증가하는 것을 확인할 수 있다. 또한, 300*300으로 연산량이 작을 때는 thread가 10일 때가 더 빠르지만, 500*500부터는 연산량이 많아지면서 thread가 100일 때가 더 빠르다. 즉, 연산량에 따라 최적의 수행시간을 얻기 위해 생성해야 할 thread의 개수가 다름을 확인할 수 있다.