Lab-3 Report

소프트웨어학과 201921085 곽수정

1. How to parallelize my game of life

- **Sequential version**

Sequential 경우에는 generation 횟수가 0보다 클 경우 이중 for문 안에서 모든 cell 각각에 대해 이웃 수를 먼저 구하게 됩니다.

하나의 cell에 대한 이웃 수를 알아내기 위해 getNumOfNeighbors 함수를 통해 neighbor에 저장하였는데, getNumOfNeighbors 함수에서는 cell의 8개의 이웃들에 대해 검사하게 됩니다. 우선 cell의 상하, 좌우 row와 col이 존재하는지를 먼저 알아보고, 존재한다면 이웃이 있다는 것이기 때문에 이웃의 상태를 isLive로 불러와 alive 상태면 neighbor에 1을 더하였으며, 이 방법으로 8개의 이웃 생존 여부를 알아내 이웃 생존 수를 return 하도록 하였습니다.

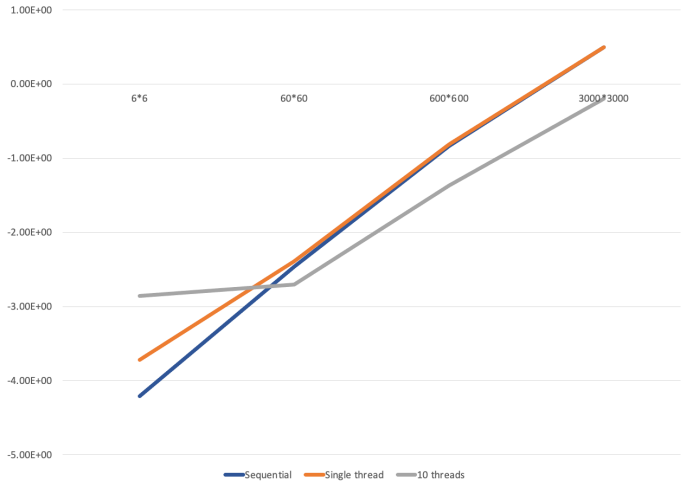
다음으로 현재 비교할 cell이 alive 상태인지, dead인지 알기 위해 isLive 함수를 호출하여 m_Grid의 정보를 불러오도록 하였습니다. 알아낸 이웃 수와 cell의 생존 정보를 바탕으로 game of live의 규칙에 따라 cell의 상태를 변경하기 위해 live 또는 dead 함수를 호출하여 m_Temp에 저장하였고, 이를 업데이트 해주기 위해 next 함수로 m_Grid에 저장하도록 하였습니다. 업데이트 후 다음 generation 으로 진행하기 위해 decGen() 함수로 generation 수를 1 감소하도록 하였습니다.

- **Single thread version & Multiple threads version**

Single thread 버전과 multiple thread 버전의 구현 방법은 동일하며, nprocs 가 1 이면 single thread, nprocs 가 2 이상이면 multiple thread 버전으로 동작합니다. 먼저 nprocs 로 입력된 값에 따라 thread id 배열을 만들고, t_args 타입의 구조체 배열을 만들도록 하였습니다. For 문에서 thread 마다 args 의 from_rows, to_rows, cols 를 초기화하였는데 thread 가 1 보다 큰 경우 입력받은 row 크기를 thread 개수로 나눠 구간마다 각각의 thread 로 parrallel 하게 실행하도록 하였습니다. 그 구간을 from_rows 와 to_rows 로 저장하였고, thread 를 생성하여 저장한 args 와 함께 workerThread 를 실행시키도록 하였습니다.

workerThread 에서는 sequential version 과 유사하게 generation 각각의 단계에서 cell 마다 이웃 수를 알아내고 이를 바탕으로 m_Temp 에 live 인지, dead 인지 저장하도록 하였습니다. 모든 thread 가 m_Temp 에 저장하는 일을 하고 나서 generation 을 1 감소시킬 수 있도록 pthread_barrier_wait 을 사용하였습니다. 성공 완료시에 barrier 에 동기화 된 thread 들 중 임의의 thread 가 PTHREAD_BARRIER_SERIAL_THREAD 를 반환하므로 이를 확인하고 generation 을 감소시키고 m_Temp 에서 m_Grid 로 업데이트 해주는 next 를 수행하도록 하였습니다. 마찬가지로 m_Grid 에 업데이트 하는 일을 모든 thread 가 완료하고 다음 generation 을 진행하기 위해 pthread_barrier_wait 으로 동기화 해주었습니다.

2. Performance comparison



size	Sequential	Single thread	10 threads
6*6	6.20E-05	0.000191	0.001388
60*60	0.003414	0.004087	0.001986
600*600	0.146351	0.153604	0.042812
3000*3000	3.1404	3.14783	0.629189

Sample_input의 simple input file에 대해 generation을 10으로 고정해두고 grid 크기를 늘려가며 sequential, single thread, 10개의 threads 버전에서 실행시간을 측정하였습니다. 왼쪽 그림은 log 그래프이고, 오른쪽은 측정시간 표입니다. 문제의 사이즈가 작을 때는 Sequential < Single thread < 10 multiple threads 로 10개의 threads 실행 경우가 가장 시간이 오래 걸렸습니다. Grid의 사이즈를 늘려 측정했을 때에는 10 multiple threads < Sequential < Single thread로 10개의 threads 실행 경우가 가장 시간이 짧게 걸렸습니다.

먼저 Sequential과 single thread 를 비교해보았을 때, sequential에서 일을 진행하는 main thread와 single thread의 thread는 같은 양의 일을 수행하지만, single thread 버전에서는 thread를 만들고, 오류없이 생성되었는지 확인하고, 일을 마친후에 join하고 이를 확인하는 등의 과정이 추가되었기 때문에 single thread가 sequential에 비해 더 오래걸리는 것을 알 수 있었습니다.

문제의 크기가 작을 때에는 다른 경우들보다 multiple thread가 더 오래걸렸지만 문제의 크기가 커질수록 5배정도 빠르게 실행되었습니다. 문제가 가장 작을 때 (6*6), 10 multiple threads의 경우 row가 6인데 thread가 10개 생성되었으므로, 다른 경우보다 thread 만드는 과정에 시간을 투자하였지만 쉬고있는 thread가 생기는 오버헤드가 생겨 sequential이나 single thread보다 더 오래걸리는 것으로 생각해볼 수 있었습니다. 문제가 가장 클 때는 (3000*3000), sequential과 single thread의 경우에는 thread 하나가 3000줄을 sequential하게 진행하였습니다. 반면 10 multiple threads의 경우 3000줄을 10개의 thread가 구간을 나눠 parallel하게 진행하였습니다. 따라서 300줄을 sequential하게 진행하는 thread 10개가 parallel하게 수행되었기 때문에 sequential과 single thread보다 5배정도 빠른 수행시간을 갖는 것을 알 수 있었습니다.

또한 위와 같이 generation 10 고정 값에서 grid 크기를 똑같이 늘려가며 5개의 thread, 10개의 thread, 20개의 thread의 실행 시간을 측정해보았습니다. 문제의 grid 크기가 작을수록 5개의 multiple thread에서의 실행시간이 짧게 걸렸고, 문제의 크기가 커질수록 20 threads에서 가장 짧은 실행시간을 가졌습니다. 이를 통해 할일이 적을 때에는 thread 생성 오버헤드를 고려하여 thread가 적은 것이 적합하고, 할일이 많아질수록 thread가 늘어나 일을 구간별로 부담하여 parallel하게 진행하는 것이 적합하다는 것을 알 수 있었습니다.