

컴퓨터정보통신공학부 10조  
2011253020 이화중  
2011253011 양성민  
2013253070 송명호

MPICH2를 이용한 행렬 곱셈

병렬처리론 MPI(MPICH2) 팀 과제

# 결과 분석

Figure 1 프로세서 개수가 8개일 때, 행렬 크기(가로 너비)에 따른 처리 시간

Figure 2 프로세서 개수가 8개일 때, 행렬 크기(원소 개수)에 따른 처리 시간

Figure 1과 Figure 2는 8대의 컴퓨터를 연결하고 행렬 크기를 바꿔가며 처리시간을 측정한 결과이다. 행렬의 크기가 일 때 계산 복잡도는, 행렬 원소 하나 당 회의 곱셈과 회의 덧셈이 필요한데 원소의 개수가 이라서 회의 연산이 필요하므로, 이다. 그러므로 연산 소요 시간은 세제곱에 비례 할 것이다. 따라서 그 결과는, 의 그래프(Figure 1의 추세선 참조)와 유사한 형태를 가질 것으로 예상된다. 즉, 작업량이 커지면 처리 시간도 그에 따라 증가한다는 것이며, 이는 Figure 1을 통해 확인할 수 있다.

Figure 2는 행렬 원소 수에 따른 시간 변화를 측정한 그래프이다. 행렬의 원소의 개수를 이라고 한다면, 전체 연산 횟수는 이다. 그런데 행렬 원소 개수가 증가하면, 전달해야 할 데이터가 늘어나므로, 통신량 또한 증가한다. 따라서, 이를 감안한다면, 계산 복잡도는 에 가까울 것으로 예상된다. Figure 2의 추세선을 통해 이를 확인할 수 있다. 이러한 결과로부터, 처리 시간은 연산 횟수에 비례한다는 것을 유추할 수 있다.

Figure 3 프로세서 개수에 따른 속도 향상

Figure 3는 배열 개수와 프로세서 개수에 따른 속도 향상을 나타낸 것이다. 대체로, 특정 개수보다 많아지면 오히려 속도 향상이 감소하는 경향을 보였다. 하나의 예시로, 의 행렬 곱셈의 속도 향상은, 프로세서가 3개일 때 그 값이 최대이다(약 1.38). 그 이후로는 오히려 감소하기 시작하여, 프로세서 개수가 7개 이후부터는 1개일 때보다도 결과가 나쁘다. 그리고 15개 이후부터는 속도 향상이 절반 이하로 떨어진다.

Minsky의 모순에 따르면, 프로세서의 개수가 증가 하더라도 정보 교환에 따른 통신 오버헤드가 발생하기 때문에, 실질적인 성능 향상은 프로세서 개수에 비례하지 못하고 로그 함수의 형태를 취한다. 또한, Amdahl의 법칙에 따르면, 병렬처리를 이용하여 얻을 수 있는 속도 향상은 병렬화 되지 못한 부분 때문에 한계가 존재한다. 프로세서 개수가 7개 이상에서의 행렬 곱셈의 결과가 1개일 때보다 나쁜 것은, 이 두 법칙을 방증하는 한 가지 예시라 할 수 있다.

더 많은 컴퓨터가 계산에 참여할수록, 전송해야 할 데이터 량도 더 많이 늘어난다. 게다가 이번 실습에서 사용한 네트워크는, 클러스터 컴퓨팅을 위한 특별한 구조가 아니라, 기존 통신망인 인터넷 네트워크를 그대로 이용한 구조이다. 이 통신망은, 다수의 계층이 서로 독립적으로 개발되어 계층 간 중복 기능이 상당히 존재하는 등, 오버헤드가 크다. 따라서 이번 실습의 결과에는 통신 오버헤드가 상당히 클 수밖에 없다.

작업의 크기가 커지더라도 병렬화 되지 않는 부분의 크기는 크게 변하지 않기 때문에, 작업의 크기를 늘려서 속도를 향상시켜, Amdahl의 법칙이 제시한 문제를 해결할 수 있다. 이에 입각하여, 행렬 크기를 4096까지 늘리면서 처리 시간을 측정하였다. 그 결과, 행렬 크기가 커질수록 속도 향상도 증가하는 경향이 있었다. 특히, 행렬의 크기가 4096일 때, 다른 크기의 결과보다도 더 선형에 가까운 속도 향상을 얻을 수 있었다. 그럼에도, 프로세서 개수가 14개일 때의 속도 향상은 약 9.5 정도에 불과했다. 이는, 행렬 원소의 개수가 프로세서 개수로 나누어 떨어지지 않아 완전하게 같은 비율로 분할되지 않은 점이나, 병렬로 수행하지 못하는 일부 연산, 네트워크 상황, 기타 통신 오버헤드 등으로 나타난 결과로 사료된다.

# 코드 분석

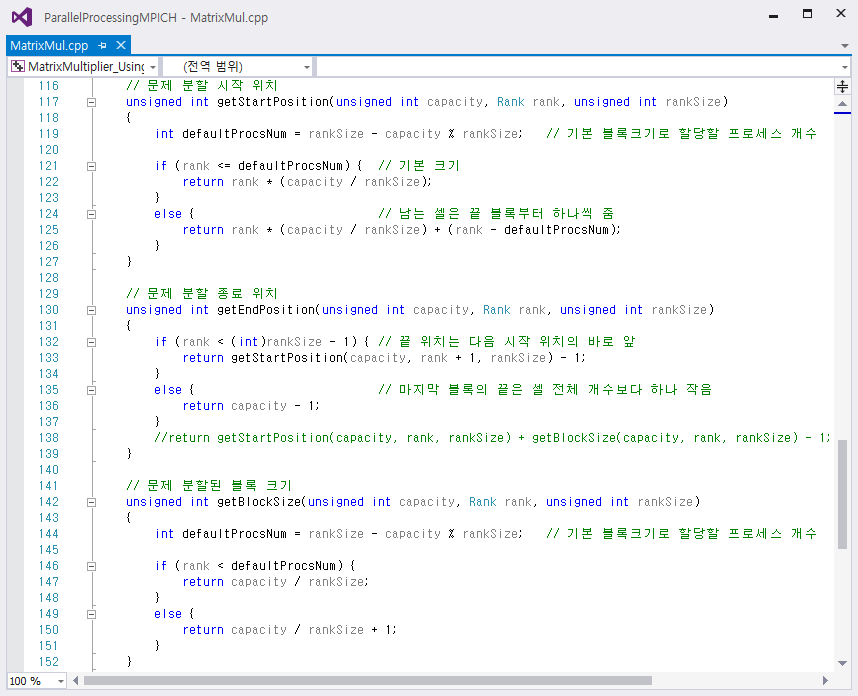


Figure 4 행렬 곱셈을 위한 문제 분할을 수행하는 함수 (MatrixMul.cpp)

병렬 처리 효과를 얻기 위해, 문제 분할하여 행렬 곱셈을 수행하였다. 실제 행렬 곱셈을 수행하기 전에 Figure 4의 함수들을 호출하여 문제를 분할한다. 문제를 분할하는 블록 크기는 기본적으로 앞행렬 원소의 개수를 프로세스의 개수로 나눈 몫이다. 하지만, 나눈 결과에 나머지가 생기는 경우, 모든 블록에 이 크기를 배정하면 어떤 프로세스도 담당하지 않는 원소가 생긴다. 그래서 남는 원소는 끝 블록부터 하나씩 추가적으로 배정한다. 따라서, 담당하는 원소의 개수가 가장 많은 프로세스와 가장 적은 프로세스의 담당 개수 차이는 최대 1개이다.

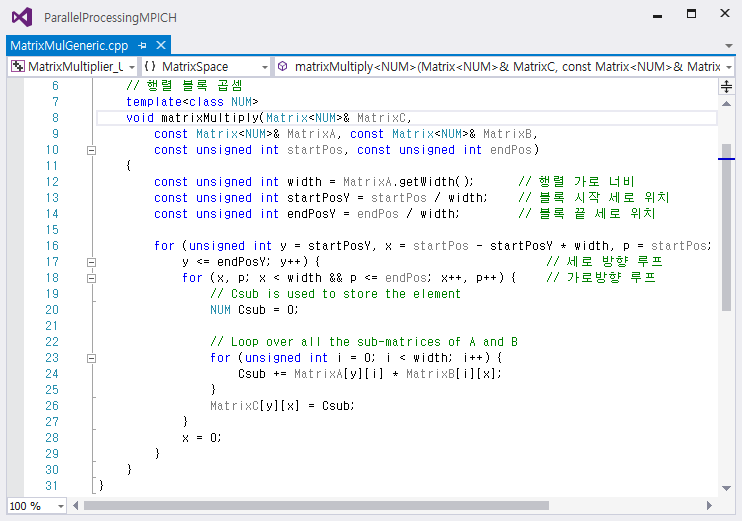


Figure 5 문제 분할된 행렬에 대한 곱셈 함수 (MatrixMulGeneric.cpp)

Figure 5의 matrixMultiply() 함수에서는, 분할된 행렬의 곱셈을 수행한다. 인자로 분할된 행렬의 시작 위치와 종료 위치를 전달받아 계산한다. 시작 위치와 종료 위치 사이의 블록 범위 내에서, 결과 행렬 원소에 해당하는 위치의 값을 각각 계산한다. 따라서 한 원소당 소요되는 곱셈의 횟수는 앞행렬(MatrixA)의 너비 또는 뒷행렬(MatrixB)의의 높이이며, 덧셈의 횟수는 이것에서 1 뺀 값이다.

아래의 Figure 6는 Host에서 수행하는 행렬 곱셈과 Satellites와의 통신을 관장하는 함수이다. 이 함수는 먼저, 값이 설정된 앞행렬 MatrixA와 뒷행렬 MatrixB를 인자로 받는다. (MatrixC는 결과를 반환할 Out-Parameter이다.) 계산 결과값을 저장할 행렬(이하. 결과 행렬)을 임시로 생성하고, 프로세스 개수에 따라 문제를 분할한다. 이때, 문제 분할은 결과 행렬을 기준으로 수행한다.

그 다음, 시간 측정을 시작(시작 시간을 기록)한 다음, 각각의 Satellites로 계산할 행렬과 문제 분할 시작 위치, 블록 크기를 전달한다. 이로써, 행렬 곱셈 수행 시간에 통신 시간을 포함한다. 전달을 완료하면 첫 번째 블록에 대하여 행렬 곱셈을 수행한다. 곱셈이 끝나면 각각의 Satellites로부터 각 블록의 결과값을 전달받아 결과 행렬에 합친다. 이제 행렬 곱셈이 완료되었으므로, 시간 측정을 종료(종료 시간을 기록)한다. 그리고 MatrixC에 결과 행렬을 복사한 다음, 측정한 시간을 반환하여 종료한다.

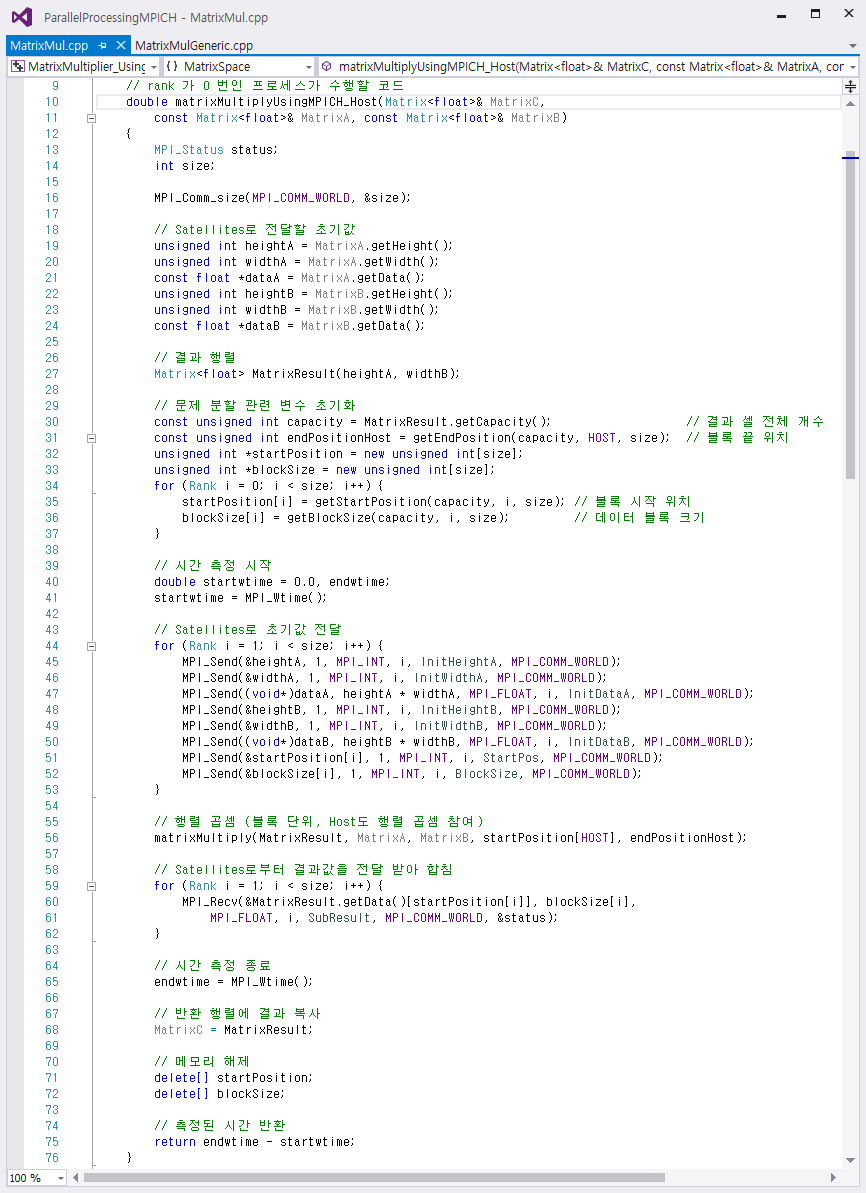


Figure 6 Host에서 수행하는 행렬 곱셈을 위한 함수 (MatrixMul.cpp)

여기서, 행렬 초기화 부분을 시간 측정에 포함하지 않았는데, 포함하는 것보다 더 적합할 것이다. Host에서는 행렬을 초기화하는 부분(Figure 8 참조)도 있으므로, 초기값 송수신부에 도달하는 시간은 각 Satellites보다 느릴 수 밖에 없다(Figure 10 참조: Host의 결과는 가장 마지막에 출력된다.). 하지만, 먼저 여기에 도달한다고 해서 행렬 곱셈을 먼저 시작하지 않으며, 값이 전달되기를 기다리고 있으므로, 모든 프로세스의 실제 행렬 곱셈은 Host로부터 Satellites에 값이 전달된 다음에 수행된다. 따라서 행렬을 초기화하는 부분을 시간 측정에 포함하지 않아도 아무 문제가 없다. 게다가 행렬을 명령행 인자나 파일로 전달받아 행렬 곱셈을 수행하는 프로그램을 작성할 수도 있다는 점도 생각하면, 이것이 더 자연스러울 것이다.

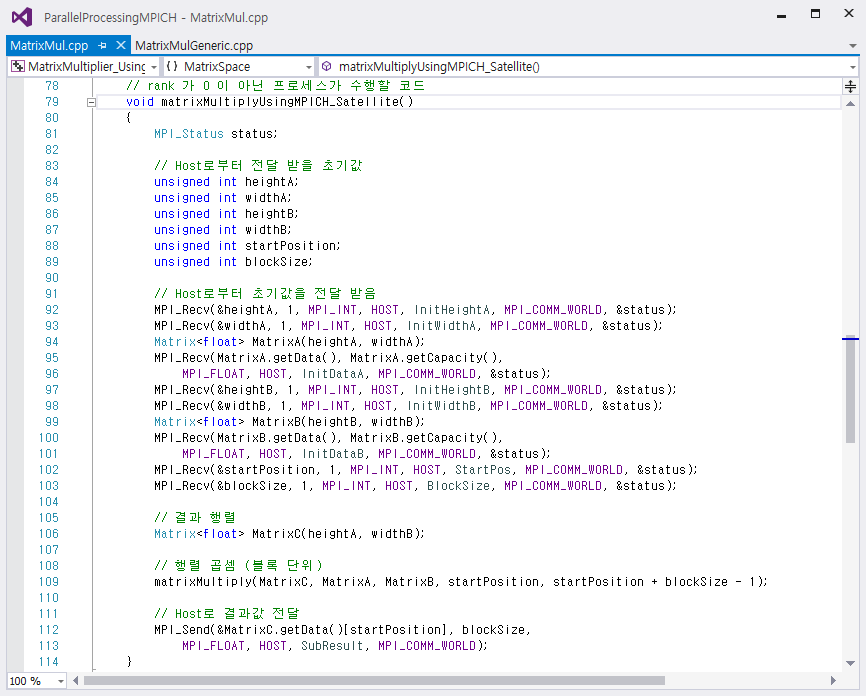


Figure 7 Satellites에서 수행하는 행렬 곱셈을 위한 함수 (MatrixMul.cpp)

Figure 7은 각 Satellite에서 수행하는 행렬 곱셈과 Host와의 통신을 관장하는 함수이다. 이 함수는 행렬 곱셈에 필요한 값들을 Host로부터 전달 받으므로, 별도의 인자를 받지 않는다. 이 함수를 시작하면, 행렬과 블록 시작 위치, 블록의 크기를 Host로부터 전달받는다. 다음으로 이 값들로 행렬 곱셈을 수행한다. 행렬 곱셈이 종료되면, 결과값을 다시 Host로 전달한다.

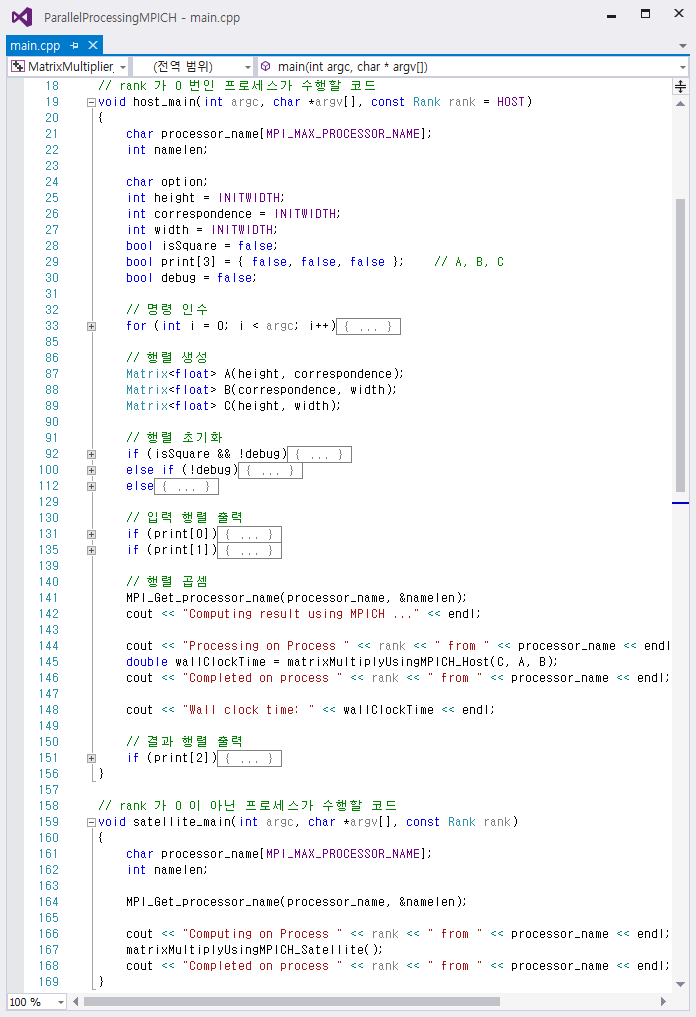


Figure 8 각 프로세스에서 수행할 메인 (main.cpp)

Figure 8의 각 함수는 각각 Host와 각 Satellites에서 수행한다. 실제 프로그램 main() 함수에서는, 수행 중인 프로세스의 번호를 가져오며, 이에 따라 host\_main() 함수를 수행할지 Satellite\_main을 수행할지 결정한다. 특별히, host\_main() 함수에는 명령행 인자를 반영하는 부분과 행렬을 초기화하는 부분을 추가하였다. 명령행 인자로 행렬 곱셈의 행렬 크기를 설정하거나 행렬을 출력할 수 있다. 사용 가능한 옵션은 아래의 Table 1에서 확인할 수 있다.

|  |  |
| --- | --- |
| 옵션 | 설명 |
| -p <matrix> | 행렬 출력 (앞행렬: a 또는 0, 뒷행렬: b 또는 1, 결과 행렬: c 또는 2 또는 임의 문자) |
| -s <size> | 계산 행렬을 정방형 행렬로, 그 높이와 너비를 설정 (-h, -c, -w 옵션을 무력화시킴) |
| -h <size> | 앞행렬과 결과 행렬의 높이 |
| -c <size> | 앞행렬의 너비와 뒷행렬의 높이 |
| -w <size> | 뒷행렬과 결과 행렬의 너비 |
| -d | 디버깅 (결과 확인이 쉽도록 앞행렬은 행, 뒷행렬은 열의 값이 일정) |

Table 1 프로그램 옵션

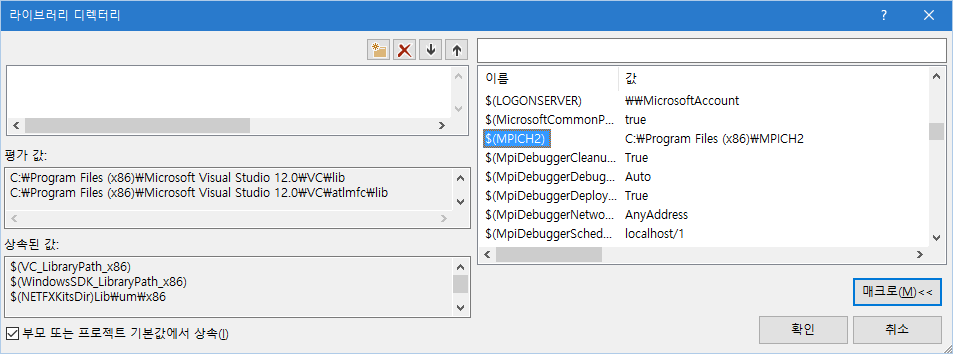


Figure 9 MPICH2 라이브러리를 사용하기 위한 환경변수 설정

MPICH2의 라이브러리를 사용하기 위하여, 프로그램에 환경변수를 도입하여 설정하였다. 따라서 이 프로그램을 빌드하기 위해서는 시스템에 환경 변수를 설정하여야 한다. 변수 이름은 “MPICH2”이며, 값은 MPICH2를 설치한 경로이다. 이 경로의 기본값은 64-bit Windows의 경우 “C:\Program Files (x86)\MPICH2”이며, 32-bit Windows의 경우는 “C:\Program Files\MPICH2”이다.

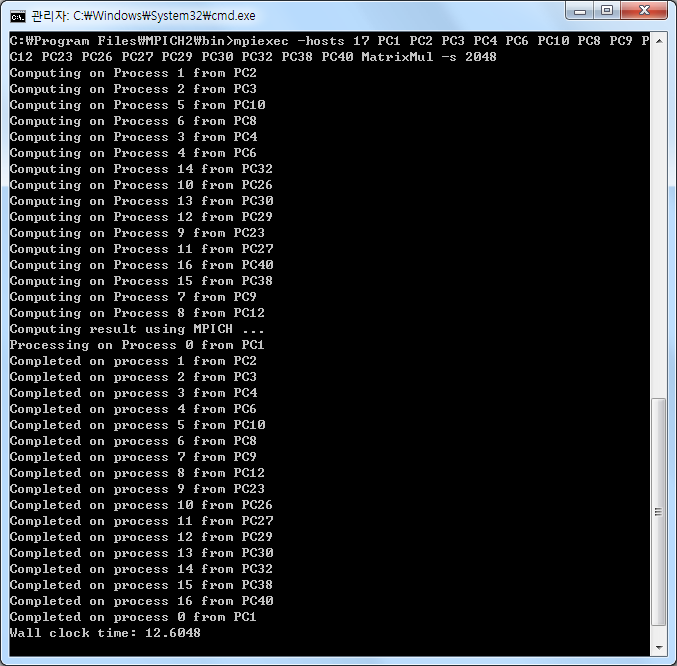


Figure 10 프로그램 실행 결과 예시

Figure 10은 이 프로그램을 실행한 결과의 예시이다. 여기서 사용한 컴퓨터는 총 17대이며, 생성한 프로세스도 17개이다. MPICH2의 수행 옵션으로 이 컴퓨터들을 설정하였다. 프로그램 실행 옵션으로는 크기가 2048인 행렬의 곱셈을 수행하도록 하였다. 더 많은 결과는 이 문서와 함께 첨부한 “2. 결과 스크린샷” 폴더에서 확인할 수 있다.

# Discussion

이번 실습에서 MPICH2를 사용하여, 다수의 컴퓨터를 클러스터링하는 방법으로 병렬 처리하여, 행렬 곱셈을 수행하였다. MPICH2를 처음 접하다 보니 생각보다 많은 어려움이 있었다.

처음 작업환경을 구성하는 부분부터 어려움이 있었는데, 제공된 자료에서 제시된 것 이상으로 더 검색하여 문제를 해결할 수 있었다. Windows 7의 경우, MPICH2를 설치할 때 명령 프롬프트를 관리자 권한으로 실행하여 명령어로 실행을 하여야 제대로 동작하였다. 하지만, Windows 10의 경우에는, 관리자 권한으로 명령어를 이용하여 설치하는 방법조차 불가능하였다.

프로그래밍할 때는, 메시지 송신 및 수신 함수에 어떤 인자를 전달해야 하는지 이해하는 데 많은 시간을 소모하였다. 특히, 프로세스 0번에서 다른 프로세스들에게 전달하는 것이나 다른 프로세스들에서 프로세스 0번으로 전달하기 위하여 어떠한 조치를 취하여야 하는지 이해하는 데 애로사항이 있었다. 그 이유는, 조원들 모두 단일 컴퓨터에서의 프로그래밍에 대한 경험 밖에 없었고, 클러스터링을 사용하는 프로그래밍이 처음이라 이해가 느렸기 때문이라 생각한다.

실습을 위하여 프로그램을 실행할 때는, 자료에 나와있는 것과 다르게 mpich2 설정에 호스트를 입력하지 않았고, 직접 인자로 전달하는 방법으로 하였다. 프로그램으로 실행하였을 때, 처음 적은 프로세스 개수로 테스트할 때는 입력한 호스트 순서대로 출력되었다. 하지만, 행렬의 크기가 커지고 프로세스의 개수가 늘어남에 따라, 입력한 호스트 순서대로가 아닌, 빠르게 수행되는 순서대로 출력되었다. 때로는, 같은 수의 프로세스를 사용하였음에도 불구하고, 가끔씩 예상에서 벗어난 시간이 출력되는 경우도 있었다. 이는, 각 호스트마다 개별적으로 실행 중인 프로그램에 따라 메모리나 CPU의 점유율이 달랐기 때문인 것으로 보인다.

속도 향상에 대하여, 호스트의 개수를 계속하여 늘리면 그와 비례하여 수행 시간이 줄어들 것으로 예상하였다. 하지만 한계가 존재했으며, 이를 기점으로 수행 시간이 오히려 증가하였다. 이를 통해, 결과 분석에서 제시한 것과 같이, Minsky의 모순과 Amdahl의 법칙을 확인할 수 있다.

실습은 창조관 큰 실습실에서 컴퓨터들을 연결하여 진행하였다. 적은 수의 컴퓨터를 사용하여도 연결율이 높은 다른 조와는 달리, 우리 조는 많은 수의 컴퓨터를 사용했음에도 연결율이 조금 저조한 편이었다. 그 이유에는 환경설정 방식이 달랐던 문제도 있었다. 하지만 더 큰 문제는, 우리 조가 사용하였던 컴퓨터를 다른 조가 사용하면서 환경설정을 바꾸다 보니, 설정이 꼬이게 되어 다시 설정해야 했다는 점이다.

실습을 진행하면서 다소 아쉬웠던 점은, 실습하는 학생들이 한번 설정된 계정을 그대로 사용하지 않고 조마다 새로 설정하여, 실습할 때마다 다시 설정해야 했기 때문에 많은 시간을 소모했다는 점이다. 그리고 다음 기회가 있다면, 이번 실습보다 더 많은 수의 컴퓨터로 더 큰 배열의 연산을 시도해보고 싶다.