creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사 학위논문

Hybrid (MPI + OpenMP) 병렬화를 통한 유동 해석 코드 병렬 최적화

Optimization on Hybrid MPI-OpenMP Parallel Program for Compressible Flow Analysis Based on Unstructured Mixed Grid

2019 년 2 월

서울대학교 대학원

기계항공공학부

김 상 혁

Hybrid (MPI + OpenMP) 병렬화를 통한 유동 해석 코드 병렬 최적화

Optimization on Hybrid MPI-OpenMP Parallel Program for Compressible Flow Analysis Based on Unstructured Mixed Grid

지도교수 김 종 암

이 논문을 공학석사 학위논문으로 제출함

2019 년 2 월

서울대학교 대학원

기계항공공학부

김 상 혁

김상혁의 공학석사 학위논문을 인준함

2019 년 2 월

위 원 장 김 규 홍

부위원장 김 종 암

위 원 이 관 중

국문초록

본 논문은 기존의 MPI 병렬기법이 적용된 혼합격자 기반 압축성 유동해석 코드에 OpenMP 병렬기법을 혼합하여 MPI와 OpenMP를 모두 사용하는 Hybrid 병렬 최적화를 수행한 연구를 다루고 있다. 유한체적법(Finite Volume Method)을 기반으로 하는 유동해석 프로그램의 경우 대부분 계산 영역을 코어 수로 나누어 계산하는 MPI 병렬 기법을 적용하고 있다. 하지만 일정 수준 이상의 많은 코어 수를 사용하면 MPI 통신 시간 비율이 증가하여 병렬 성능이 감소하는데 이를 개선하기 위해 OpenMP 병렬 기법을 추가로 적용하여 Hybrid 프로그램을 만들었다. 그 과정에서 공유 메모리에 복수의 쓰레드가 동시에 저장(write)할 때 발생하는 경합 조건(race condition)을 확인하고 개선하였으며 OpenMP 스케쥴링이나 시스템에 따른 쓰레드 배치 등 Hybrid 병렬 최적화 방법들에 대해 기술하였다. 또한 Hybrid 프로그램으로 다양한 격자와 조건에서 테스트하며 기존 MPI 코드 대비 병렬 성능 및 확장성이 향상된 것을 확인하였다.

주요어 : 전산유체역학, OpenMP, MPI, 하이브리드 병렬 프로그래밍, Speedup, 병렬 확장성
학 번 : 2017-26436

목 차

초록 ····· i
목차 ····· ii
표 목차 ····· iv
그림 목차 ····· v
제 1 장 서 론 ····· 1
1.1 병렬 프로그래밍 ····· 1
1.1.1 MPI ····· 2
1.1.2 OpenMP ····· 3
1.1.3 Hybrid ····· 4
1.2 연구 목표 ····· 5

제 2 장 지배방정식 및 유한체적법 ····· 6
2.1 지배방정식 ····· 6
2.1.1 이상 기체 방정식 ····· 7
2.2 유한체적법 ····· 8

제 3 장 Hybrid 프로그래밍 최적화 ····· 9
3.1 경합 조건(Race Condition) ····· 9
3.1.1 Residual 계산 ····· 11
3.1.2 이상 기체 상태방정식 ····· 14
3.2 OpenMP 스케쥴링 ····· 16
3.3 쓰레드 배치 ····· 17

제 4 장 수치 실험 ····· 19
4.1 실험 방법 ····· 19
4.1.1 Speedup ····· 19
4.1.2 Scalability ····· 20
4.1.3 시간 측정 구간 ····· 20
4.1.4 계산 시스템 ····· 21
4.1.5 해석 대상 및 조건 ····· 23
4.2 실험 결과 ····· 25
4.2.1 형상에 따른 병렬 확장성 ····· 25
4.2.2 차원에 따른 병렬 확장성 ····· 27
4.2.3 유동에 따른 병렬 확장성 ····· 28
4.2.4 격자수에 따른 병렬 확장성 ····· 29
4.2.5 Hybrid 조합에 따른 병렬 확장성 ····· 31
4.2.6 노드 내 필요 메모리 ····· 31

제 5 장 결 론 ····· 34

참고문헌 ····· 35
Abstract ····· 37

표 목 차

표 4.1 수치 실험을 수행한 계산 시스템 ····· 21
표 4.2 해석 형상 및 격자 ····· 24
표 4.3 노드 내에서 코어 수에 따른 필요 메모리 ····· 33

그 림 목 차

그림 1.1 CPU 제조사 제품별 clock frequency ····· 1
그림 1.2 MPI, OpenMP, Hybrid 병렬 기법 개념도 ····· 2
그림 3.1 경합 조건 (Race Condtion) 예시 ····· 10
그림 3.2 Residual 계산에 필요한 격자의 구성 요소 ····· 11
그림 3.3 기존 Residual 계산 의사코드 ····· 12
그림 3.4 개선 Residusl 계산 의사코드 ····· 12
그림 3.5 기존 이상 기체 상태방정식 개략도 ····· 14
그림 3.6 개선 이상 기체 상태방정식 개략도 ····· 15
그림 3.7 OpenMP 스케쥴링 종류 ····· 16
그림 3.8 SMP와 NUMA ····· 17
그림 4.1 Intel KNL7250 Chip 구조 ····· 22
그림 4.2 CRM 양력계수 그래프 ····· 23
그림 4.3 CRM 항력계수 그래프 ····· 23
그림 4.4 형상에 따른 병렬 확장성 ····· 25
그림 4.5 VST의 차원에 따른 병렬 확장성 ····· 27
그림 4.6 RAE2822의 유동 종류에 따른 병렬 확장성 ····· 28
그림 4.7 CRM의 유동 종류에 따른 병렬 확장성 ····· 28
그림 4.8 2차원 VST의 격자수에 따른 병렬 확장성 ····· 30
그림 4.9 3차원 VST의 격자수에 따른 병렬 확장성 ····· 30
그림 4.10 CRM의 격자수에 따른 병렬 확장성 ····· 30
그림 4.11 MPI, OpenMP 조합에 따른 성능 ····· 31
그림 4.12 노드 내에서 코어 수에 따른 필요 메모리 ····· 33

제 1 장 서 론

1.1 병렬 프로그래밍

20세기 후반부터 21세기 초반까지 매년 평균 50%씩 증가하던 싱글 프로세서(single processor)의 성능은 그림 1.1과 같이 기술과 비용의 한계로 약 4.0Ghz에서 정체되었다. 싱글 프로세서의 성능 증가는 전자회로의 밀집도가 높을수록 증가하는 트랜지스터의 속도에 기인하며, 이에 따라 전력소모가 증가하고 대부분 열로 발산되는데 이를 일반적인 공랭식으로 식히기에는 한계가 있었기 때문이다. 따라서 마이크로프로세서의 주요 제조사들은 이를 해결하기 위해 싱글 프로세서의 성능을 증가시키기보다 한 기판 위에 여러 개의 프로세서를 배치하는 방향으로 프로세서를 설계하기 시작하였으며 이를 멀티코어 프로세서라고 한다 [1].

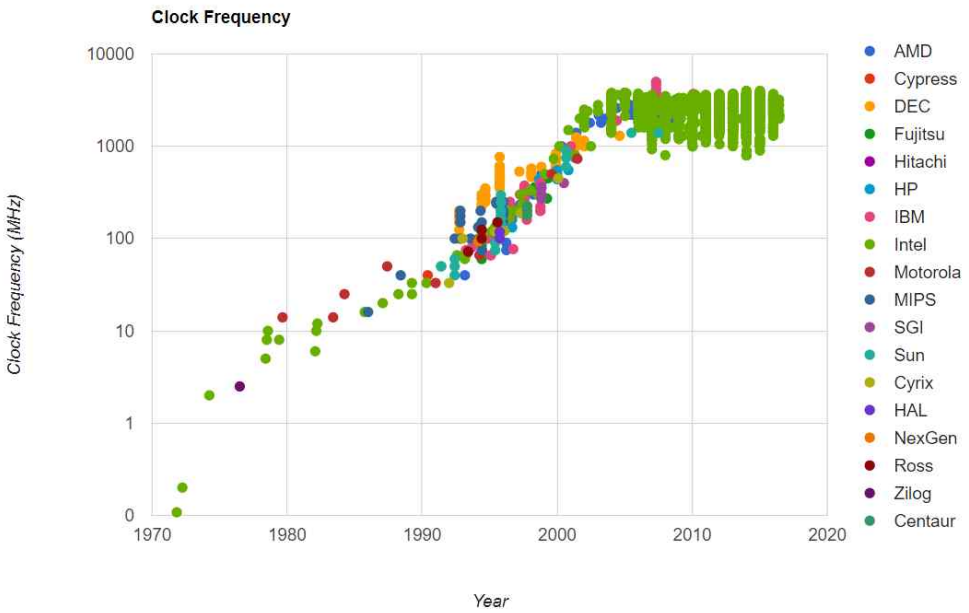


그림 1.1 CPU 제조사 제품별 clock frequency [15]

이러한 추세에 따라 기존 싱글 프로세서에 맞추어 작성되던 시리얼(serial) 프로그램들도 멀티코어를 활용할 수 있도록 병렬 프로그램으로의 수정이 불가피하게 되었다. 그 과정에서 먼저 다양한 병렬 기법 중에 사용할 프로그램 성격과 계산 환경에 적절한 병렬 기법을 선택해야 하며, 시리얼 프로그램에서는 발생하지 않았지만 병렬화로 인해 발생하는 문제들을 해결하고, 여러 최적화를 거쳐야만 2개 이상의 코어들을 효율적으로 활용하는 병렬 프로그램을 작성할 수 있게 되었다.

본 논문에서 사용한 프로그램은 다양한 병렬 프로그래밍 기법들 중 일반적으로 많이 사용되는 Message Passing Interface(MPI)와 Open Multi-Processing(OpenMP)이 적용되었다.

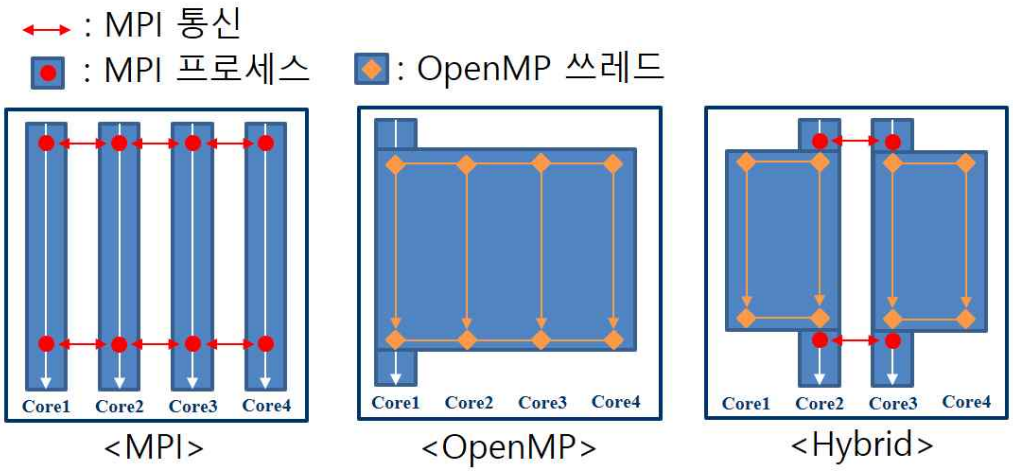


그림 1.2 MPI, OpenMP, Hybrid 병렬 기법 개념도

1.1.1 MPI

MPI 병렬 기법은 그림 1.2 왼쪽과 같이 자신들의 메모리를 따로 가지는 프로세스(process)들이 서로 필요한 데이터를 메시지로 만들어 네트워크를 통해 송신(send), 수신(receive)하여 통신하는 방법으로, 주로 프

로그램 실행 초기에 처리해야 할 전체 일을 각 프로세스에게 나누어 분배하여 실행 시간을 줄여서 병렬화한다. MPI 병렬 기법에 관련된 더 자세한 내용은 참고문헌[1, 12]을 참고한다.

이러한 MPI 병렬 기법은 분산 메모리(distributed memory) 시스템 또는 공유 메모리(shared memory) 시스템 등 다양한 플랫폼에서 구현이 가능하다는 장점이 있다. 그러나 기본적으로 각 프로세스들이 통신선을 통해 데이터를 주고받기 때문에, 사용하는 코어 수가 증가할수록 통신에 필요한 버퍼 메모리(buffer memory)의 증가 및 집합 통신 시 배리어(barrier)에 의한 프로세스의 대기 시간 증가 등 통신 관련 부하도 점차 증가하게 된다. 만약 이러한 통신 관련 부하가 프로그램 실행 초기에 영역을 분할하여 계산 시간을 줄여주는 MPI 병렬 기법의 이점보다 커지게 되면 어느 시점부터는 병렬 성능이 감소하여 오히려 코어 수가 증가할수록 프로그램 실행 시간도 더 증가하게 된다.

1.1.2 OpenMP

OpenMP 병렬 기법은 그림 1.2 가운데와 같이 병렬 구간(주로 반복문)마다 쓰레드(thread)를 생성하고 이후엔 소멸시키며 병렬화하는 방법이다. 여기서 쓰레드란 프로세스 내에서 생성되는 객체로, 동일 프로세스 내의 쓰레드들은 동일한 메모리를 공유하나 스텍(stack)과 레지스터(register)는 각각 독립적으로 가지는 실행 주체이다. OpenMP 병렬 기법에 관련된 더 자세한 내용은 참고문헌[1, 13]을 참고한다.

이러한 OpenMP 병렬 기법은 MPI 병렬 기법과 다르게 통신선 사용하지 않고 복수의 쓰레드가 공유 메모리에 접근하기 때문에 통신 관련 부하 없이 병렬화 할 수 있으며 쓰레드 개수 증가에 따른 사용 메모리 증가량이 크지 않다는 장점이 있다. 또한 점진적으로 병렬화가 가능하며 내장된 스케줄링(scheduling) 옵션으로 각 쓰레드에게 일을 배분해주기

때문에 로드 밸런싱(load balancing)이 상대적으로 용이하다. 그러나 메모리를 공유하는 시스템 내에서만 사용이 가능하기 때문에 병렬 확장성이 떨어지며, 직렬 프로그램이나 MPI 병렬 기법에서는 코어마다 독립된 메모리만을 사용하기 때문에 경험할 수 없었던 경합조건(race condition)과 같은 오류들이 발생한다는 단점이 있다.

1.1.3 Hybrid

Hybrid 병렬 기법은 MPI와 OpenMP 병렬 기법 각각의 단점을 보완하고 장점을 조합한 방법이다. 기본 개념은 그림1.2 오른쪽과 같이 MPI로 통신하는 프로세스마다 OpenMP 쓰레드를 생성하면서, 메모리를 공유하는 노드 내에서는 OpenMP와 MPI의 조합으로, 메모리를 공유하지 않는 노드 간은 MPI로 병렬화 하는 것이다. 그 결과 많은 코어를 사용하여 프로그램 실행 시 MPI만 사용했을 경우보다 상대적으로 MPI 프로세스 수를 적게 사용하기 때문에 통신 관련 부하는 감소하고 MPI 프로세스 수 감소로 부족한 병렬 성능은 OpenMP 쓰레드로 보충하므로 향상된 병렬 성능을 기대할 수 있다. 또한 메모리적인 측면에서도 MPI만으로 이루어진 프로그램보다 Hybrid가 더 유리하다. Hybrid 병렬기법 관련 문법 등 더 자세한 내용은 참고문헌[2, 3]을 참고한다.

그러나 이러한 이점들을 얻기 위해서는 두 가지 병렬 기법들의 특성뿐만 아니라 프로그램 완성 이후 활용할 시스템 구조를 이해하고 그에 맞게 코어들을 적절히 배치해야 한다.

1.2 연구 목표

유한체적법(Finite Volume Method, FVM)을 기반으로 하는 유동 해석 프로그램에서는 저장해야할 데이터가 격자 개수에 비례하기 때문에, 격자수가 수천만 개 이상인 대규모 해석에서 요구되는 메모리는 매우 크다. 그러므로 대규모 해석을 고려하는 유동 해석 프로그램들은 주로 많은 코어를 활용할 수 있도록 확장성이 좋고 전체 계산 영역을 코어 수로 분할하여 메모리를 나눌 수 있는 MPI 병렬 기법을 적용하고 있으며 소속 연구실에서 보유한 In-house 압축성 유동 해석 코드에도 기존에 MPI가 적용되어 있었다. 그러나 1.1.1절에서 언급하였듯이 통신을 기반으로 하는 MPI 병렬 기법 특성상 많은 코어를 활용할 경우 통신 관련 부하들을 무시할 수 없게 되어 성능이 감소한다.

따라서 본 연구의 목표는 많은 코어를 활용 시 기존 MPI 병렬 기법이 적용된 혼합격자 기반 압축성 유동 해석 프로그램의 병렬 성능 향상을 위해 OpenMP 병렬 기법이 추가로 적용된 Hybrid 프로그램을 개발하고 기존 MPI 프로그램과 성능을 비교, 분석하는 것이다.

제 2 장 지배방정식 및 유한체적법

2.1 지배방정식

점성과 열 전도의 영향을 고려한 지배방정식인 나비에-스토크스(Navier-Stokes) 방정식은 다음과 같다.

$$\frac{\partial \boldsymbol{U}}{\partial t} + \nabla \cdot (\boldsymbol{F_c} - \boldsymbol{F_v}) = 0 \tag{2.1}$$

여기서 t는 시간이며, $\boldsymbol{U}$는 보존 변수 벡터, $\boldsymbol{F_c}$는 비점성 플럭스 텐서, $\boldsymbol{F_v}$는 점성 플럭스 텐서로 3차원에서 각각 아래와 같다.

$$\boldsymbol{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e_t \end{bmatrix}, \ \boldsymbol{F_c} = \begin{bmatrix} \rho u & \rho v & \rho w \\ \rho u^2 + p & \rho uv & \rho uw \\ \rho uv & \rho v^2 + p & \rho vw \\ \rho uw & \rho vw & \rho w^2 + p \\ \rho uH & \rho vH & \rho wH \end{bmatrix}, \ \boldsymbol{F_v} = \begin{bmatrix} 0 & 0 & 0 \\ \tau_{xx} & \tau_{yx} & \tau_{zx} \\ \tau_{xy} & \tau_{yy} & \tau_{zy} \\ \tau_{xz} & \tau_{yz} & \tau_{zz} \\ \theta_x & \theta_y & \theta_z \end{bmatrix} \tag{2.2}$$

여기서 ρ, p, e_t, H와 (u, v, w)는 각각 밀도, 압력, 전에너지(total energy), 전엔탈피(total enthalpy)와 속도 벡터이다. 또한 τ_{ij}는 점성 플럭스 텐서로 뉴튼 유체(Newtonian fluid)를 가정할 때 스토크스 가설(Stokes' hypothesis)에 의해 식 (2.3)으로 표현되며, $\theta_x, \theta_y, \theta_z$는 식 (2.4)와 같이 정의된다.

$$\tau_{ij} = \mu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3}\frac{\partial u_k}{\partial x_k}\delta_{ij}\right) \tag{2.3}$$

$$\begin{aligned}\theta_x &= u\tau_{xx} + v\tau_{xy} + w\tau_{xz} + q_x \\ \theta_y &= u\tau_{yx} + v\tau_{yy} + w\tau_{yz} + q_y \\ \theta_z &= u\tau_{zx} + v\tau_{zy} + w\tau_{zz} + q_z\end{aligned} \tag{2.4}$$

이 때 q_i는 열 플럭스로 푸리에 법칙(Fourier's law)에 의해 식 (2.5)와 같이 표현된다.

$$q_i = -k\frac{\partial T}{\partial x_i} \tag{2.5}$$

여기서 μ는 점성 계수, k는 열전도계수, T는 온도를 의미한다.

2.1.1 이상 기체 상태방정식

이상 기체 상태방정식은 이상 기체(ideal gas)의 상태 변수 값을 구하기 위해 사용되며 식 (2.6)과 같다. 또한 정적 비열 c_p와 정압 비열 c_v가 일정한 열량적 완전기체(calorically perfect gas)를 가정했을 때 식 (2.7)을 만족한다.

$$p = \rho RT \tag{2.6}$$

$$\begin{aligned}e &= c_v T \\ h &= c_p T\end{aligned} \tag{2.7}$$

여기서 p는 압력, ρ는 밀도, T는 온도, e는 내부에너지, h는 엔탈피, R은 기체상수 $8.31\,J/mol \cdot K$이다.

이 때 식 3개에 상수를 제외한 상태변수는 5개이므로 만약 2개의 상태변수 값이 주어지면 나머지 상태변수의 값을 구할 수 있음을 알 수 있다.

2.2 유한체적법

본 연구에서 사용한 프로그램은 혼합격자 기반 압축성 유동 해석 프로그램으로 비점성과 점성, 층류와 난류, 아음속과 초음속 등 다양한 유동에 대해 해석이 가능하다. 지배방정식인 식 (2.1)을 유한체적법(Finite Volume Method)으로 표현하면 검사 체적(control volume) Ω에 대해 아래와 같이 정리된다.

$$\frac{\partial}{\partial t}\int_{\Omega} \boldsymbol{U}\, d\Omega + \int_{\partial\Omega} [(\boldsymbol{F_c} - \boldsymbol{F_v}) \cdot \boldsymbol{n}]\, dS = 0 \tag{2.8}$$

여기서 $\boldsymbol{n} = [n_x n_y n_z]$은 바깥 방향 수직 벡터(normal vector), 각 검사 체적에서의 보존 변수 벡터 $\boldsymbol{U}$와 대류 플럭스 텐서 $\boldsymbol{F_c}$, 점성 플럭스 텐서 $\boldsymbol{F_v}$는 식 (2.2)와 동일하다.

이 때 식 (2.8)을 반차분(semi-discrete) 형태로 다시 표현하면 다음과 같다.

$$\Omega \frac{\partial \overline{U}}{\partial t} = -\sum_{\mathrm{if}ace} (Flux[\mathrm{if}ace] \cdot S[\mathrm{if}ace]) \tag{2.9}$$

여기서 Ω는 체적, $\overline{U}$는 검사 체적 평균(cell-averaged) 보존 변수 벡터이며, $\mathrm{if}ace$는 검사 체적에 인접한 표면들, $Flux[\mathrm{if}ace]$는 $\mathrm{if}ace$에서 계산된 플럭스 벡터 $[(\boldsymbol{F_c} - \boldsymbol{F_v}) \cdot \boldsymbol{n}]$, $S[\mathrm{if}ace]$는 $\mathrm{if}ace$의 면적 크기이다.

제 3 장 Hybrid 프로그래밍 최적화

MPI 병렬 기법은 분산 메모리 시스템을 기반으로 개발된 기법이기 때문에 MPI로 병렬화 된 프로그램은 각 코어가 자신만의 독립된 메모리를 가지고 계산을 수행한다. 그러므로 다른 코어와 통신하는 부분을 제외하면 시리얼 프로그램과 동일하게 작성해도 무방하다. 하지만 OpenMP 병렬 기법의 경우 공유 메모리 시스템 기반 병렬 기법이기 때문에 복수의 쓰레드가 메모리를 공유할 때 추가적인 문제들이 발생할 가능성이 생기며, 적절하게 처리하지 않을 경우 의도하지 않은 결과가 랜덤하게 나타나기 때문에 이를 바로 잡는 과정에서 시리얼 프로그램보다 더 많은 비용을 요구하게 된다. 뿐만 아니라 MPI와 OpenMP를 혼합한 Hybrid 프로그램을 사용할 경우 계산 시스템에 따라 적절한 옵션을 사용하지 않으면 성능이 크게 달라질 수 있다.

따라서 본 장에서는 OpenMP를 적용하는 과정에서 주의해야 할 오류 현상과 Hybrid 프로그램 완성 이후 계산 시스템에 따라 고려해야할 사항들에 대해 다루고 있다.

3.1 경합 조건 (Race Condition)

경합 조건[9]은 넓은 의미로 비동기화된 동시다발적 실행으로 인해 예측할 수 없는 결과를 유발하는 프로그래밍 오류를 뜻하며 좁은 의미로는 복수의 쓰레드가 동일한 메모리에 무질서하게 접근하며 새로운 데이터를 공유 메모리에 쓰려고 할 때 발생하는 오류인 데이터 경합 조건을 의미한다. 본 논문에서는 후자의 의미로 사용한다.

이러한 경합 조건은 OpenMP 뿐만 아니라 Pthread와 같은 공유 메모리 기반 병렬 기법에서 자주 발생하는 현상이다.

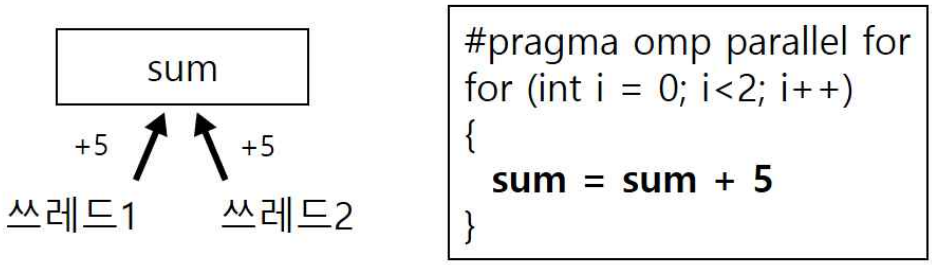


그림 3.1 경합 조건 (Race Condition) 예시

예를 들어 그림 3.1과 같이 공유 변수 sum에 5를 두 번 더해주는 반복문에 대해 OpenMP 쓰레드 2개를 생성하여 반복문을 병렬로 수행하면 경합 조건이 발생하는데, 결론적으로 sum에 5가 더해질지 10이 더해질지 예측할 수 없다. 이는 쓰레드1이 자신의 레지스터로 변수 sum을 로드(Load)하고 5를 더하는 시점에서 쓰레드2가 자신의 레지스터로 sum을 로드한다면, 쓰레드2가 로드한 값은 쓰레드1이 5를 더한 값이 아니라 그 전의 초기 값이기 때문이다.

또한 공유 변수에 접근하는 쓰레드의 순서는 규칙성이 없으므로 어떠한 경우에는 쓰레드1의 “로드-계산-저장” 과정이 끝난 뒤에 쓰레드2가 “로드-계산-저장”을 하면 본래 의도한 대로 10이 더해진 값을 출력할 수도 있다. 이렇게 확률적으로 정상적인 값을 출력할 수도 있으며 컴파일 단계에서 거를 수 있는 오류가 아니기 때문에 사전에 경합조건을 정확히 예측하지 못한다면 오류를 찾고 수정 하는 것에 많은 시간이 소요된다.

다음은 기존 MPI 프로그램에 OpenMP를 적용하는 과정에서 확인한 경합 조건이 발생한 부분들과 실제적인 해결 방안에 대해 다루고 있다.

3.1.1 Residual 계산

Residual이란 지배방정식인 식 (2.1)인 나비에-스토크스 방정식을 Semi-discrete한 식 (2.9)에서 시간 관련된 좌측항을 제외한 나머지 우측항을 의미한다.

$$\Omega\frac{\partial \overline{U}}{\partial t} = -\sum_{\mathrm{if}ace}(Flux[\mathrm{if}ace] \cdot S[\mathrm{if}ace]) \tag{2.9}$$

여기서 Ω는 셀의 부피, $\overline{U}$는 셀 평균(Cell-averaged) 보존 변수 벡터이며, $\mathrm{if}ace$는 셀에 인접한 표면들, $Flux[\mathrm{if}ace]$는 $\mathrm{if}ace$에서 계산된 플럭스 벡터 $[(\boldsymbol{F_c}-\boldsymbol{F_v}) \cdot \boldsymbol{n}]$, $S[\mathrm{if}ace]$는 $\mathrm{if}ace$의 면적 크기이다.

이 때 중요한 점은 Flux는 face마다 계산되는 값인 반면, Residual은 각 cell마다 계산되는 값이라는 것이다. 또한 그림 3.2에서 알 수 있듯이 바깥 방향 노말 벡터 n_f의 방향에 따라 owner cell과 neighbor cell이 구분되어 Flux 값이 인접 cell의 Residual에 더할지 뺄지가 결정된다는 것이다.

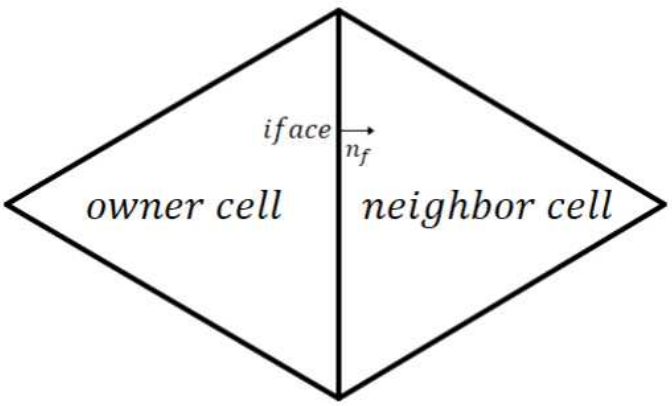


그림 3.2 Residual 계산에 필요한 격자의 구성 요소

```
for iface = 1 ~ total face
{
  Flux[iface] = H(owner cell, neighbor cell, n_f, ...)
  Residual[owner cell]      += Flux[iface] × S[iface]
  Residual[neighbor cell]  −= Flux[iface] × S[iface]
}
```

그림 3.3 기존 Residual 계산 의사코드

```
#pragma omp parallel for schedule(guided)
for iface = 1 ~ total face
{
  Flux[iface] = H(owner cell, neighbor cell, ...)
}

#pragma omp parallel for schedule(guided)
for icell = 1 ~ total cell
{
  for iface = adjacent face of icell
  {
    if (icell==owner cell of iface)
      Residual[icell]  += Flux[iface] × S[iface]
    else
      Residual[icell]  −= Flux[iface] × S[iface]
  }
}
```

그림 3.4 개선 Residual 계산 의사코드

이를 바탕으로 기존 프로그램에 구현되어 있던 Residual을 계산하는 부분을 의사코드(Pseudo-code)로 표현하면 그림 3.3과 같다. 전체 계산 영역의 모든 face에 대해서, 인접한 셀들의 물리량을 가지고 계산한 Flux에 face 면적을 곱한 뒤 바로 owner cell의 Residual에는 더해주고 neighbor cell의 Residual에는 빼주었다.

이러한 알고리즘을 직렬로 실행하거나 MPI로 각 코어마다 할당된 영역을 독립적으로 계산하게 할 때는 문제가 발생하지 않는다. 하지만 OpenMP로 위의 for문을 병렬화할 경우 3.1절에서 언급한 경합 조건이 발생한다. 이는 한 cell에 인접한 face가 여러 개가 있기 때문에 face를 돌며 cell을 업데이트 할 경우 복수의 쓰레드가 동일한 cell의 Residual을 업데이트할 가능성이 있기 때문이다.

따라서 이러한 가능성을 제거하기 위해 기존에 face 반복문만 사용하던 것을 그림 3.4와 같이 face 반복문과 cell 반복문으로 구분하였다. 먼저 face 반복문에서는 기존과 같이 각 face를 돌며 Flux를 계산 후 저장한다. 이 때는 각 쓰레드가 할당받는 face는 서로 중복되지 않으므로 저장하는 Flux 값도 중복되지 않는다. 이후 cell 반복문에서는 cell을 기준으로 하였을 때, 인접한 face들의 관점에서 자신이 owner cell인지 neighbor cell인지 판단하여 그 face의 Flux 값을 자신의 cell의 Residual에 반영하게 하였다. 그러면 OpenMP로 병렬화 하였을 때 복수의 쓰레드가 특정 cell의 Residual에 동시에 저장할 가능성이 없어지므로 직렬로 실행했을 때와 동일한 결과를 얻으면서 경합조건을 피할 수 있다.

3.1.2 이상 기체 상태방정식

2.1.1절에서 기술하였듯이 이상 기체 상태방정식으로 식 (2.6)을, 열량적 완전 기체를 가정하였을 때 식 (2.7)을 사용할 수 있다.

$$p = \rho R T \tag{2.6}$$

$$\begin{aligned} e &= c_v T \\ h &= c_p T \end{aligned} \tag{2.7}$$

본 연구에 사용한 프로그램의 언어는 C++며 기존 프로그램에 구현된 이상 기체 상태방정식은 아래와 같다.

그림 3.5 기존 이상 기체 상태방정식 개략도

여기서 가운데 사각형 박스는 5개의 상태변수가 멤버 변수로 정의된 이상 기체 상태방정식의 객체를 의미한다.

예를 들어 그림과 같이 상태 변수 중 밀도 ρ와 에너지 e를 객체로 입력받았을 때 함수를 통해 식 (2.6)과 식 (2.7)을 계산하여 객체 내에 멤버 변수로 저장되었다가 나머지 압력 p, 온도 T, 엔탈피 h 중에 필요한 멤버 변수 값을 반환한다.

이러한 알고리즘을 직렬로 실행하거나 MPI로 각 코어마다 할당된 영역을 독립적으로 계산하게 할 때는 문제가 발생하지 않는다. 하지만 위의 알고리즘을 수정 없이 OpenMP로 병렬화할 경우, 상태 변수인 멤버 변수는 객체가 생성될 때마다 모든 쓰레드가 공유하는 변수이므로 그림 3.5에서 계산 및 저장하는 단계에서 여러 쓰레드가 동시에 저장하려 할 때 경합 조건이 발생한다.

따라서 이를 해결하기 위해 먼저 상태 변수들을 멤버 변수에서 모두 제거하고 멤버 함수 내에서만 존재하도록 임시적으로 변수로 선언하였다. 그 뒤에 두 값을 인자로 입력받아 선언한 값에 대입하고 각 식으로 계산한 뒤 그 결과를 바로 반환할 수 있도록 하였다. 그러면 각 쓰레드가 동일한 멤버 함수를 호출하더라도 계산에 필요한 상태 변수의 메모리는 각 쓰레드마다 다른 위치에 개별적으로 저장되기 때문에 동시에 동일한 메모리에 접근하는 가능성이 없어져서 경합 조건을 피할 수 있다.

그림 3.6 개선 이상 기체 상태방정식 개략도

3.2 OpenMP 스케쥴링

OpenMP 병렬 기법은 주로 반복문을 병렬화하는데 여러 쓰레드가 일을 처리하는 방식은 옵션에 따라 다르며 프로그램 성격에 따라 성능에도 영향을 미친다. 아래 그림은 대표적인 스케줄링 옵션 3가지에 대해 다루고 있다.

첫 번째는 static으로 전체 일의 양을 모든 쓰레드가 균등한 개수로 나누어 처리한다. 각각의 일 로드가 비슷할 때 유리한 옵션이다. 두 번째로 dynamic은 일정 수준의 일을 동일한 크기로 나누어 놓고 처리가 먼저 끝나는 쓰레드들에게 나눠주는 방법이다. 쓰레드들이 작업이 끝났는지 매번 확인해야하므로 상대적으로 부하가 큰 방법이다. 마지막으로 guided는 dynamic과 비슷하지만 처음엔 큰 크기로 나누어 일하다가 점점 일의 크기를 줄여가는 방법이다.

반복문의 성격에 따라 가장 성능이 좋은 옵션을 찾아내야 하는데 본 연구에 사용한 프로그램의 경우 guided가 가장 성능이 좋았으며 그 다음으로 static, dynamic 순으로 성능이 좋았다.

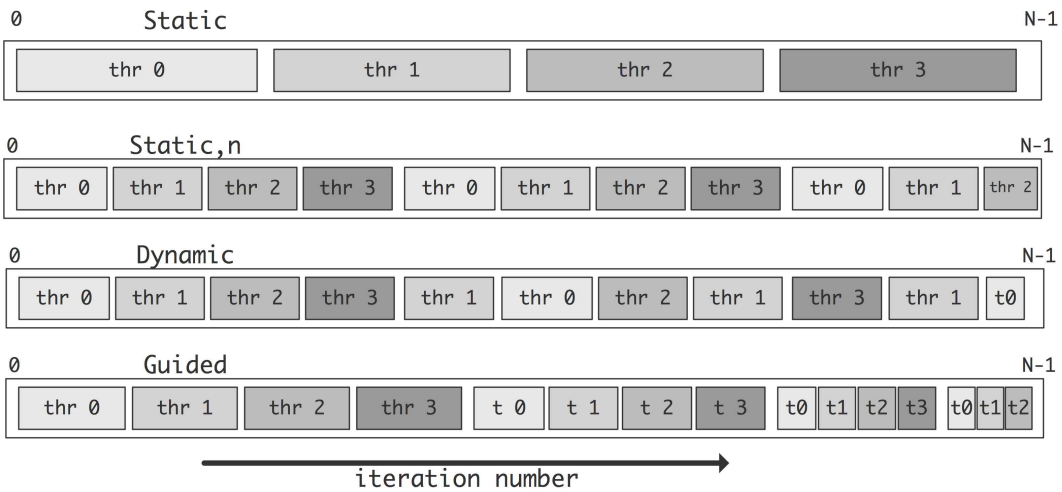


그림 3.7 OpenMP 스케쥴링 종류

3.3 쓰레드 배치

프로세스마다 독립된 메모리를 가지는 MPI와 달리 OpenMP 쓰레드가 포함된 Hybrid는 프로그램 실행 전에 반드시 시스템 구조를 고려하여 프로세스와 쓰레드의 배치를 확인해야 한다. 같은 프로그램을 실행해도 쓰레드를 적절하게 배치하지 않으면 성능에 큰 손실을 입게 된다.

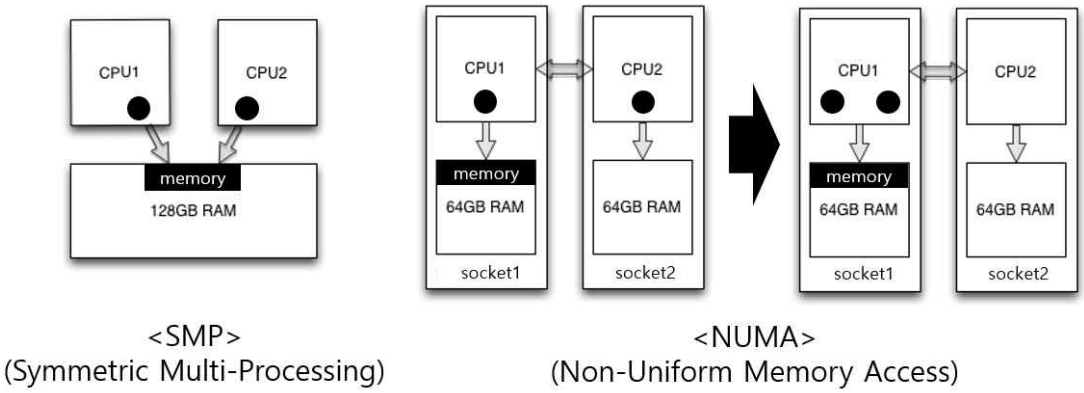


그림 3.8 SMP와 NUMA

다중 처리 시스템은 한 시스템에 둘 이상의 중앙처리장치(CPU)를 이용하여 병렬 처리할 수 있는 시스템으로 크게 대칭형 다중 처리(Symmetric Multi-Processing, SMP)와 불균일 기억 장치 접근(Non-Uniform Memory Acess, NUMA) 두 종류로 나눌 수 있다. SMP는 둘 이상의 CPU가 한 개의 공유된 메모리를 사용하기 때문에 어느 CPU에서 메모리에 접근해도 걸리는 시간은 동일하다. 하지만 NUMA에서는 각 CPU마다 자신의 메모리를 가지고 있기 때문에 자신과 먼 메모리에 접근할 때가 자신과 가까운 메모리에 접근할 때보다 훨씬 느리다.

만약 OpenMP 프로그램을 사용할 때 그림 3.7에서 왼쪽과 같이 SMP 시스템이라면 쓰레드가 어느 CPU에 배치되든 메모리에 접근하는 시간이 비슷하기 때문에 크게 신경 쓰지 않아도 된다. 하지만 NUMA 시스

템일 경우 메모리는 CPU1과 가까운 곳에 할당이 되었는데 그림 3.7의 가운데 그림처럼 쓰레드가 CPU1과 CPU2에 각각 한 개씩 배치된다면 두 쓰레드의 매번 메모리 접근 시간에 차이가 생기고 이는 동기화가 필요한 병렬 프로그램 특성상 성능 저하로 이어질 수 밖에 없다. 따라서 그림 3.7의 오른쪽과 같이 가능한 쓰레드들을 동일한 CPU 또는 동일한 소켓 내에 배치되도록 해야 하는데 이는 OpenMP 병렬 기법의 환경변수인 OMP_PLACE와 OMP_PROC_BIND를 적절하게 설정함으로써 가능하다.

또한 Hybrid 프로그램에서는 OpenMP 쓰레드 뿐만 아니라 MPI 프로세스의 배치도 성능에 영향을 미치는 요인이다. 가장 이상적인 상황은 각 MPI 프로세스 내에서 생성된 OpenMP 쓰레드들은 서로 가깝게, MPI 프로세스들은 중간에 OpenMP 쓰레드들이 들어갈 만큼 충분한 간격을 가지고 배치되어야 한다. 이를 위해 MPI의 라이브러리와 버전에 맞는 옵션을 별도로 설정해주어야 한다. 예를 들어 Intel MPI를 사용할 경우에는 환경변수 I_MPI_PIN_DOMAIN를 적절히 설정해주면 된다.

본 연구의 실험 시스템인 4호기의 경우 1노드에 2소켓, 소켓당 4코어이므로 MPI 프로세스와 OpenMP 쓰레드의 가능한 조합은 MPI 2개와 OpenMP 4개 또는 MPI 4개와 OpenMP 2개가 가능하다.

제 4 장 수치 실험

4.1 실험 방법

4.1.1 Speedup

병렬 컴퓨팅 분야에서 Speedup은 속도가 향상된 정도를 뜻하며 아래와 같이 표현된다.

$$S = \frac{T_s}{T_p} \tag{4.1}$$

$$T_p = T_{compute} + T_{communicate} \tag{4.2}$$

여기서 S는 Speedup, T_s는 직렬(serial)로 계산한 시간, T_p는 병렬(parallel)로 계산한 시간이며 T_p는 아래와 같이 $T_{communicate}$는 병렬 계산 시간 T_p 중에 MPI 통신에 사용한 시간, $T_{compute}$는 통신을 제외한 나머지(연산, 메모리 할당 등)에 사용한 시간이다.

식 (4.1)에서 일반적으로 T_s는 코어 1개, T_p는 코어 2개 이상을 사용하여 계산한 시간을 의미한다. 하지만 1개 코어로는 실행이 되지 않을 경우(ex. 대용량 계산 시 메모리 제한)나 한 노드의 코어를 꽉 채우지 않으면 활성 코어에 전력을 몰아주어 성능이 향상시키는 인텔 터보 부스트 기능에 의해 결과가 정확하지 않다고 판단될 경우 등에서 직렬의 기준은 한 노드의 코어 개수가 될 수 있다. 식 (4.2)는 병렬로 계산한 시간 T_p를 MPI 관련 루틴에 사용된 시간 $T_{communicate}$과 그 외의 계산 시간 $T_{compute}$로 구분한 식이다.

4.1.2 Scalability

병렬 컴퓨팅 분야에서 Scalability는 병렬 확장성을 뜻하며 크게 Strong Scalability와 Weak Scalability 두 가지가 있다. Strong Scalability는 전체 문제 크기를 고정하고 사용 코어 수를 늘려가면서 걸리는 시간을 기준으로 한다면 Weak Scalability는 코어 당 문제 크기를 동일하게 유지하여 코어 수를 늘려갈 때 걸리는 시간을 기준으로 한다.

본 연구의 수치 실험은 Strong Scalability를 기준으로 진행되었다.

4.1.3 시간 측정 구간

연구에 사용한 유동해석 프로그램의 경우 크게 세 부분으로 이루어져 있다. 첫 번째는 계산할 격자 정보 및 조건들을 불러오고 MPI 랭크 수에 맞게 도메인을 나눈 뒤 각 도메인에서 계산에 필요한 메모리를 할당하는 전처리(Pre-processing) 과정이다. 이 후 두 번째는 목표하는 조건을 만족할 때까지 계산을 반복하는 유동해석(Flow analysis) 과정이며, 마지막은 최종 결과 데이터들을 파일로 출력(Post)하는 후처리(Post-processing) 과정이다.

본 연구에서 측정하는 구간은 두 번째인 유동해석 과정이다. 첫 번째 전처리 과정과 마지막 후처리 과정은 프로그램이 실행되는 동안 오직 한 번만 실행되는 반면, 유동해석 과정은 목표하는 조건에 따라 반복 계산되며 일반적으로 전체 실행시간 중 가장 큰 비율을 차지하기 때문이다. 또한 입출력(I/O) 관련 병렬 확장성은 계산(Computation) 관련 병렬 확장성과는 또 다른 이슈이기 때문에 유동해석 과정 내에도 중간 계산 결과를 출력하는 부분을 제외하고 오직 계산하는 시간만을 측정하도록 설정하였다.

4.1.4 계산 시스템

같은 프로그램이라 하더라도 계산 환경에 따라 성능은 크게 달라질 수 있다. 결과의 객관성 및 신뢰성 확보를 위해 아래 표와 같이 한국과학기술정보연구원(Korea Institute of Science and Technology Information, KISTI)의 두 시스템 Tachyon2(이하 4호기)와 Nurion(이하 5호기)을 사용하였으며 자세한 스펙은 아래 표에 나타내었다.

시스템	Tachyon2 (4호기)	Nurion (5호기)
프로세서	Intel Xeon X5570 (Nehalem)	Intel Xeon Phi 7250 (KNL)
기본 주파수	2.93GHz	1.40GHz
노드당 코어 수	4 * 2 socket	68
노드당 메모리	24GB DDR3-1333	16GB DDR4-2400 * 6 Channel
하이퍼쓰레드	Off	Off
인터커넥트	Infiniband QDR	Intel OPA
컴파일러	GCC 4.9.3	Intel 18.0.3
MPI ver	Openmpi 1.8.5	Intel mpi 18.0.3
OpenMP ver	4.0	4.5

표 4.1 수치 실험을 수행한 계산 시스템

두 시스템의 가장 큰 차이점은 노드당 코어 수이다. 4호기는 노드당 코어 4개로 이루어진 프로세서 2개, 총 8코어로 이루어져 있다. 반면에 5호기는 그림 4.1과 같이 코어 2개로 이루어진 타일이 34개, 한 노드에 총 68개의 코어를 가지고 있다. 단, 본 연구에서는 OS 관련 로드를 배제하기 위해 총 68코어 중 첫 번째 타일과 마지막 타일의 코어 4개를 제외한 64코어만을 사용하였다. 또한 두 시스템 모두 하이퍼쓰레드[4]는 비활성화 되어 있으므로 코어당 쓰레드는 1개씩 사용하였다.

또한 3.3절에서 설명하였듯이 Hybrid 병렬 프로그램은 시스템에 따라 한 노드 내에서 MPI 프로세스와 OpenMP 쓰레드 조합이 성능에 중요하게 작용한다. 본 연구에서는 4호기의 경우 1노드(8코어)에서 소켓이 2개이므로 가장 성능이 좋았던 MPI 프로세스 2개와 각 소켓당 OpenMP 쓰레드 4개를 사용하였으며, 5호기의 경우 1노드(64코어)에서 MPI 프로세스 16개와 OpenMP 쓰레드 4개를 사용하였다.

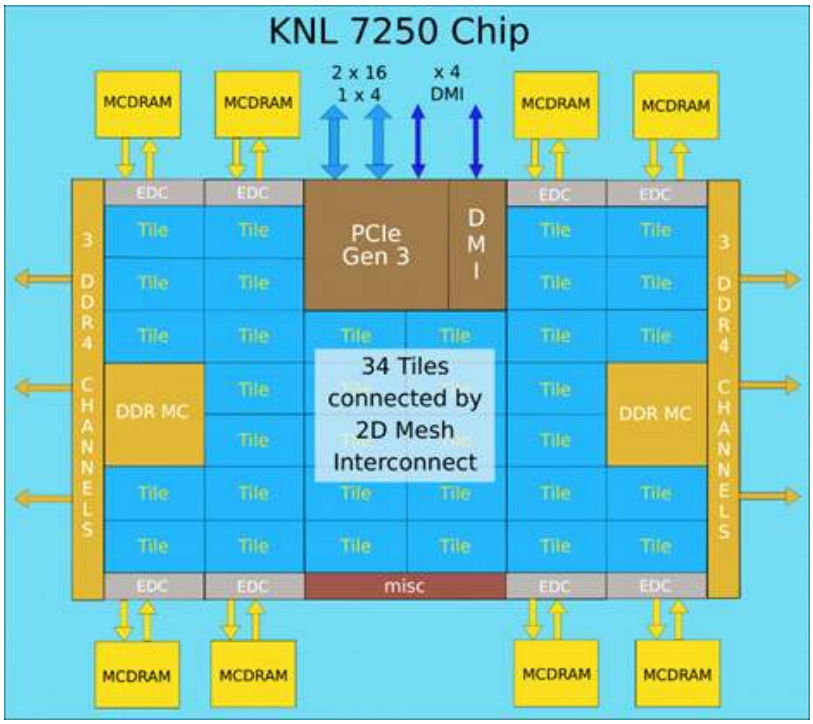


그림 4.1 Intel KNL 7250 Chip 구조 [14]

4.1.5 해석 대상 및 조건

성능 실험에 사용한 해석 대상으로 형상 및 격자는 표 4.2에 정리하였으며 각각에 대한 물리적인 해석 조건은 동일 표의 종류에 표시한 각 레퍼런스를 참고한다. 해석에 사용한 수치기법으로 시간 차분법은 외재적(Explicit) 3차 정확도 TVD Runge-Kutta 방법[5]을 사용하였다. 수치플럭스 함수로는 유한체적법에서 주로 사용되는 Roe's Flux Difference Splitting(FDS)를 개선한 RoeM 기법[6]을 사용하였으며 충격파와 같은 불연속 영역의 수치진동을 제어하기 위해 다차원 공간 제한 기법(MLP)[7]을 적용하였다.

본 연구에서는 병렬 성능을 확인하는 것이 주 목적이므로 각 문제에 대해 실제 실험값과 시뮬레이션 결과를 비교하는 것보다 MPI로 해석한 결과와 Hybrid로 해석한 결과를 비교하는 것에 중점을 두었다. 특히 실험에 사용한 시간 차분법이 외재적(Explicit) 방법이기 때문에 MPI와 Hybrid의 해석 결과는 동일한 MPI 프로세스 수를 사용할 경우 완전히 동일해야 한다. 예를 들어 표 4.2에서 CRM에 대하여 1024코어를 사용하여 MPI와 Hybrid로 각각 계산하였을 때 CL, CD 그래프는 아래와 같이 동일하다.

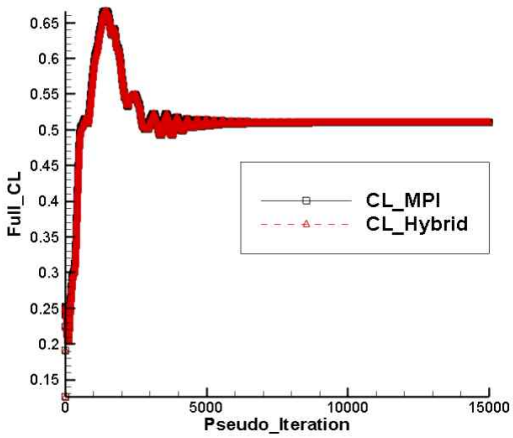


그림 4.2 CRM 양력계수 그래프

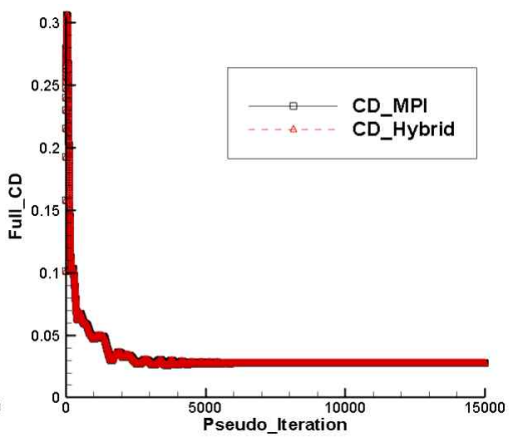


그림 4.3 CRM 항력계수 그래프

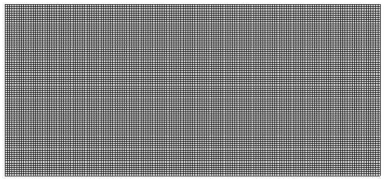
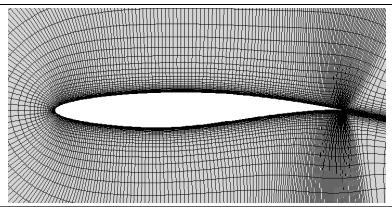
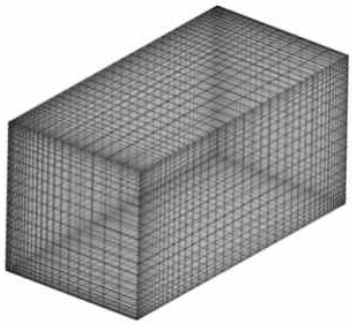
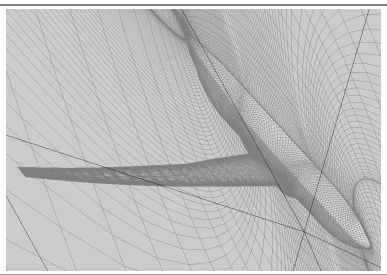
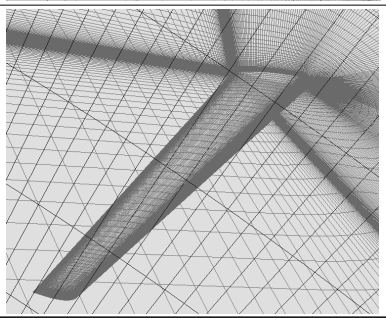
차원	종류	격자	유동 정보
2D	Viscous Shock Tube (VST)		$(\rho,u,v,p)_L$ $=(120,0,0,120/\gamma)$ $(\rho,u,v,p)_R$ $=(1.2,0,0,1.2/\gamma)$
	RAE2822		Re 6.2×10^6 $Mach$ 0.729 AOA 2.31
3D	Viscous Shock Tube (VST) [8]		$(\rho,u,v,w,p)_L$ $=(120,0,0,0,120/\gamma)$ $(\rho,u,v,w,p)_R$ $=(1.2,0,0,0,1.2/\gamma)$
	Common Research Model (CRM) [10]		Re 5.1×10^6 $Mach$ 0.85 AOA 2.3
	DPW-Wing [11]		Re 4.0×10^6 $Mach$ 0.76 AOA 0.5

표 4.2 해석 형상 및 격자

4.2 실험 결과

4.1절의 내용을 기반으로 수치 실험을 수행하였다. 대부분 Speedup을 지표로 MPI와 Hybrid의 병렬 성능을 비교, 분석하였으며 후반부에 노드 내 조합과 메모리 관련한 분석도 추가로 진행하였다.

4.2.1 형상에 따른 병렬 확장성

차원	형상	격자수	시스템
2D	Viscous Shock Tube	23,544	4호기
2D	RAE2822	23,552	4호기

본 실험은 물리적인 형상이 병렬 확장성에 영향이 있는지 알아보기 위해 진행하였다. 형상은 2차원 Viscous Shock Tube인 직사각형과 2차원 날개 RAE2822이며, 격자수는 각각 23,544개와 23,552개인 약 0.03%의 차이로 동일 수준으로 가정하였다.

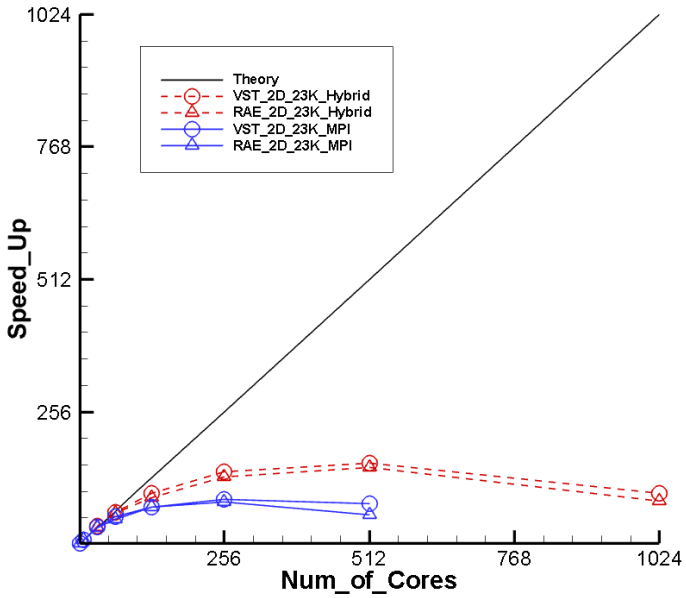


그림 4.4 형상에 따른 병렬 확장성

4호기에서 실험 결과 그림 4.4와 같이 병렬 확장성은 두 형상이 거의 동일하게 나타났다. 이는 동일 차원, 동일 격자수일 경우 물리적으로 형상이 다르더라도 프로그램 내 계산 루틴은 동일하며 병렬 확장성을 결정하는데 주요 요인이 아님을 알 수 있다.

또한 MPI는 512코어 수준까지 확장되며 Hybrid는 1024코어 이상의 수준으로 확장이 가능함을 확인할 수 있다. 이는 프로그램과 4호기 시스템 특성상 허용하는 MPI 프로세스 수가 약 512개인데 Hybrid는 MPI 프로세스를 전체 코어수의 1/4만 사용하고 나머지는 OpenMP 쓰레드로 대체하기 때문이다. 만약 이와 같이 어떤 시스템에 사용 가능한 MPI 프로세스 수가 제한되어 있다면 Hybrid의 OpenMP 쓰레드를 추가로 사용함으로서 더 많은 코어를 활용할 수 있다.

한편 본 실험에 있어서 MPI와 Hybrid 모두 이론적인 성능에 크게 미치지 못하는 이유는 코어당 계산해야할 문제 크기(격자수)가 작기 때문이다. 예를 들어 256코어를 사용할 경우 MPI는 한 코어당 반복문마다 약 92개의 내부 셀을 계산하게 되는데 이로 인해 식 (4.2)에서 계산 시간 $T_{compute}$이 통신 시간 $T_{communicate}$이 보다 매우 작게 되기 때문이다.

4.2.2 차원에 따른 병렬 확장성

차원	형상	격자수	시스템
2D	Viscous Shock Tube	2,000,000	4호기
3D	Viscous Shock Tube	2,000,000	4호기

본 실험에서는 Viscous Shock Tube 형상을 동일한 유동조건, 동일한 격자수에 대해 차원만 다르게 하여 병렬 확장성을 비교하였다.

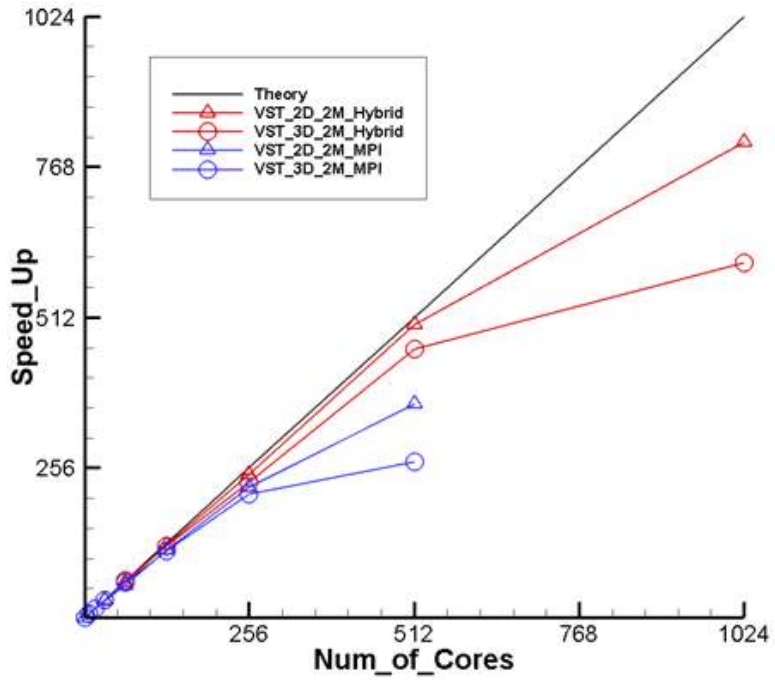


그림 4.5 VST의 차원에 따른 병렬 확장성

위 그림을 보면 3D보다 2D가, MPI보다 Hybrid가 병렬 성능이 더 좋다는 것을 확인할 수 있다.

3D보다 2D가 병렬 성능이 더 좋은 이유는 전체 셀 개수가 동일하다고 했을 때 셀 1개당 표면 수는 3D가 6개로 4개인 2D보다 1.5배가 많기 때문이다. 그러므로 3D가 face를 따라 MPI 통신하는 시간이 더 많기 때문에 식 (4.2)의 $T_{communicate}$의 비율이 증가하여 2D의 병렬 성능이 떨어지게 된다.

4.2.3 유동에 따른 병렬 확장성

차원	형상	격자수	시스템
2D	RAE2822	23,552	4호기
3D	Common Research Model	1,277,952	5호기

본 실험에서는 RAE2822와 CRM 형상에 대해 비점성 유동과 난류 유동(난류 모델 SA, SST)을 각각 적용하여 해석을 진행하며 병렬 확장성을 비교하였다.

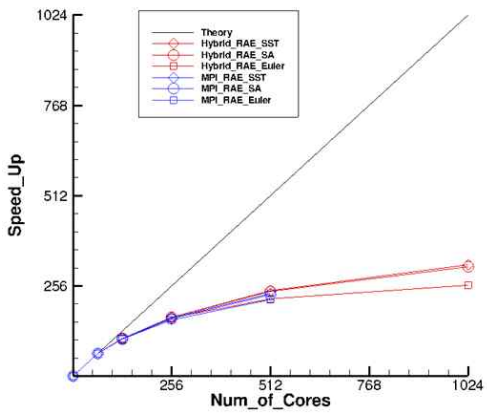


그림 4.6 RAE2822의 유동 종류에 따른 병렬 확장성

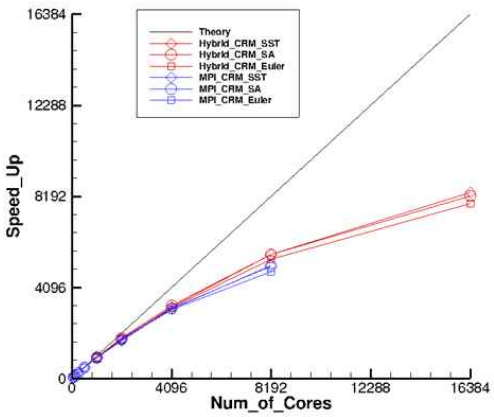


그림 4.7 CRM의 유동 종류에 따른 병렬 확장성

그림 4.6와 4.7을 보면 Euler < SA < SST 순으로 병렬 성능이 좋다는 것을 확인할 수 있다. 이는 비점성 유동의 경우 각 셀마다 $\rho, \rho u, \rho v, \rho w, \rho E$ 총 5개 변수를 가지고 계산하지만 1-Equation 모델인 SA는 변수 Nu tilde를, 2-Equation 모델인 SST는 변수 Turbulent Kinetic Energy와 Turbulent Dissipation Rate를 추가적으로 더 계산하기 때문에 계산량이 증가하여 식 (4.2)의 $T_{compute}$의 비율이 증가하기 때문이다.

4.2.4 격자수에 따른 병렬 확장성

차원	형상	격자수	시스템
2D	Viscous Shock Tube	12,800 23,544 2,000,000	4호기
3D	Viscous Shock Tube	23,276 840,000 2,000,000	4호기
3D	Common Research Model	638,976 1,277,952 2,981,888	5호기

본 실험에서는 위의 표와 같이 각 형상과 동일한 유동조건에 대해 격자수만 다르게 하며 병렬 확장성을 비교하였다.

그림 4.8 ~ 4.10를 보면 MPI보다 Hybrid가, 격자수가 많을수록 병렬 성능이 더 좋다는 것을 확인할 수 있는데 이는 식 (4.2)로 분석하면 다음과 같다.

T_p를 구성하는 $T_{communicate}$과 $T_{compute}$를 병렬 성능 측면에서 보면, $T_{communicate}$는 각 코어 사이에 통신하거나(점대점 통신) 또는 모든 코어들이 동시에 통신하며(집합 통신) 서로 가진 데이터를 동기화하는 함수들이 포함되어있기 때문에 메시지를 주고받는 동안 대기하는 시간이 필연적으로 발생하게 된다. 따라서 코어수가 증가할 때 $T_{communicate}$가 정비례해서 줄어들지는 않으며, 많은 코어를 사용하여 동기화할 경우 오히려 대기 시간이 급격히 증가하게 된다. 이에 비해 $T_{compute}$는 연산에 관련된 시간이며 코어수와 비례해서 분할된 영역만을 각 코어가 계산하기 때문에 상대적으로 병렬 성능이 더 좋다.

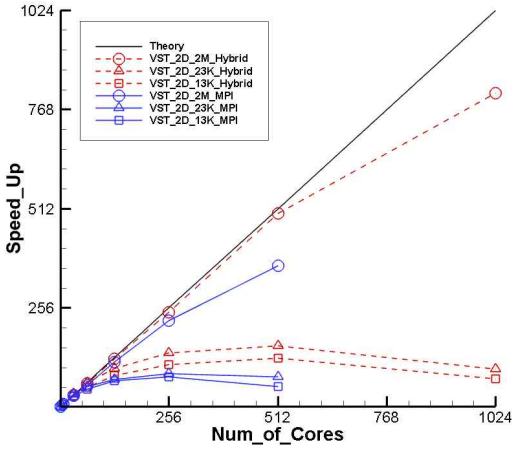


그림 4.8 2차원 VST의 격자수에 따른 병렬 확장성

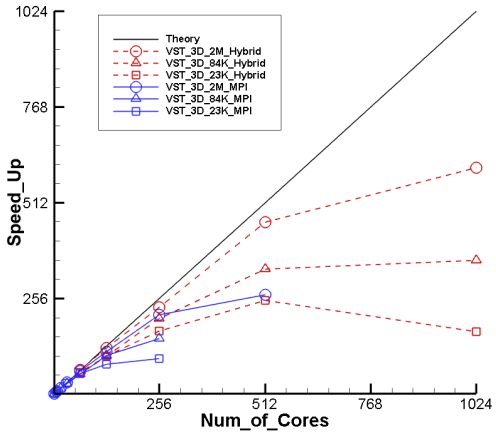


그림 4.9 3차원 VST의 격자수에 따른 병렬 확장성

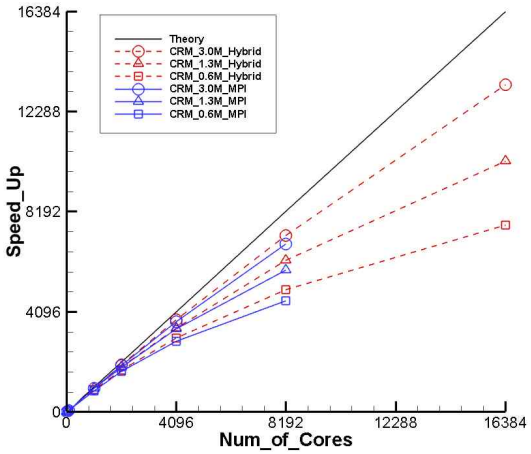


그림 4.10 CRM의 격자수에 따른 병렬 확장성

4.2.5 Hybrid 조합에 따른 병렬 확장성

차원	형상	격자수	시스템
3D	DPW-Wing	8,331,264	5호기

본 실험에서는 각 노드당 64개의 코어를 가지는 5호기에서 MPI 프로세스와 OpenMP 쓰레드 조합 비율에 따른 성능을 분석하였다. 앞의 표와 같이 격자수 8,331,264개인 3차원 DPW-Wing을 대상으로 노드 32개(총 1024코어)를 사용하여 계산하였을 때 그림 4.11과 같이 MPI 프로세스 16개, OpenMP 쓰레드 4개 조합이 가장 좋은 성능을 보였다.

이는 OpenMP로 병렬화할 때 병렬 구간마다 모든 쓰레드를 생성시키고 소멸시키는 부하와, 반복문이 끝날 때마다 존재하는 암시적 배리어(implicit barrier)가 존재하기 때문에 무조건 쓰레드 수가 많다고 좋은 것이 아니라는 것을 알 수 있다. 쓰레드 수가 너무 많을수록 쓰레드 생성/소멸시키며 동기화해야하기 때문이다. 반대로 MPI 프로세스가 가장 많을 경우 통신 부하로 인해 성능이 감소한다. 따라서 프로그램과 시스템에 따라 가장 좋은 성능을 보이는 최적의 조합을 찾아야 한다.

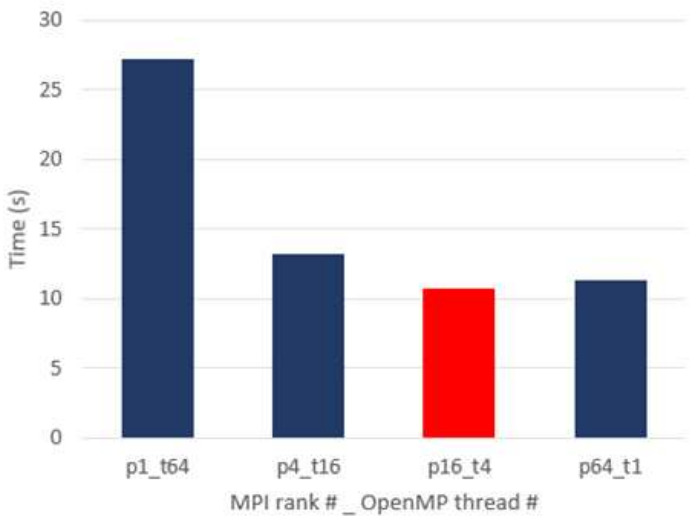


그림 4.11 MPI, OpenMP 조합에 따른 성능

4.2.6 노드 내 필요 메모리

차원	형상	격자수	시스템
3D	Common Research Model	638,976	5호기

본 실험에서는 5호기 1노드에서 코어 개수를 1개부터 64개까지 2배수로 증가시키며 프로그램 실행에 필요한 노드 내 최대 메모리 크기를 비교, 분석하였다. 위의 표와 같이 격자수 638,976개인 3차원 CRM 형상에 대해 Euler로 해석한 결과를 표 4.3과 그림 4.12에 나타내었다. 이 때 4.1.4절에서 언급하였듯이 Hybrid는 주어진 코어수의 1/4를 MPI 프로세스로 사용하므로 Hybrid의 최소 코어 개수는 4개이다.

실험 결과 64코어를 사용하였을 때 Hybrid가 MPI보다 필요 메모리를 약 36% 절약할 수 있음을 확인하였다. MPI 병렬 기법은 계산 이전에 전체 영역의 셀 개수를 코어 수로 나누어 각 코어가 분할된 영역을 맡아 해석을 진행하는 동시에, 인접한 영역의 정보를 알기 위해 필연적으로 경계면에 메모리를 추가로 할당해야하는데 이를 MPI 셀이라 한다. 이로 인해 코어수를 많이 사용할수록 프로그램에 필요한 메모리도 증가하게 되며, 한 노드의 최대 메모리에 따라 해석할 수 있는 문제의 최대 크기가 제한된다.

예를 들어 표 4.3의 MPI에서 코어 수가 1개일 때 필요한 메모리는 2.4492GB이며 이 때 MPI 셀은 없다. 만약 코어 개수가 증가하면 MPI 셀 개수도 증가하며 전체 메모리도 증가하는데 5호기의 1노드에서 64코어를 사용하였을 때 필요한 메모리는 4.9874GB로 1코어일 때보다 약 2배의 메모리가 요구됨을 알 수 있다.

한편 OpenMP 병렬 기법은 병렬화가 필요한 구간에서만 지역변수를 임시적으로 생성했다가 구간이 끝나면 바로 소멸시키므로 쓰레드를 증가시킬 때 프로그램에 필요한 최대 메모리 증가량은 상대적으로 매우 작다. 예를 들어 MPI에서 MPI 프로세스 수를 16개에서 64개로 4배 증가시킬 때 메모리 증가량은 $\frac{4.9874-3.1745}{3.1745}\times 100 \fallingdotseq 57.11\%$인 반면, Hybrid의 경우 동일한 MPI 프로세스 16개에서 각각 OpenMP 쓰레드 4개를 생성하

여 64개 코어를 사용할 때 메모리 증가량은 $\frac{3.2014-3.1745}{3.1745}\times 100 \fallingdotseq 0.85\%$이다. 그러므로 1노드 내에서 Hybrid의 n 코어를 사용할 때 필요한 메모리는 MPI에서 n/4개의 코어를 활용할 때 사용되는 메모리 수준임을 알 수 있다.

이러한 장점은 CFD 분야에서 1억 개 이상의 격자수를 가지는 대규모 문제나 adjoint variable method 같이 각 셀에 식 (2.2)의 보존 변수들 외에 더 많은 변수 값들을 저장해야하는 디자인 기법을 적용할 때 Hybrid 기법이 메모리 한계를 극복하는 좋은 해법이 될 수 있음을 알 수 있다.

Number of cores	MPI (GB)	Hybrid (GB)
1	2.4492	-
2	2.5554	-
4	2.6326	2.4501
8	2.8319	2.5571
16	3.1745	2.6401
32	3.5824	2.8427
64	4.9874	3.2014

표 4.3 노드 내에서 코어 수에 따른 필요 메모리

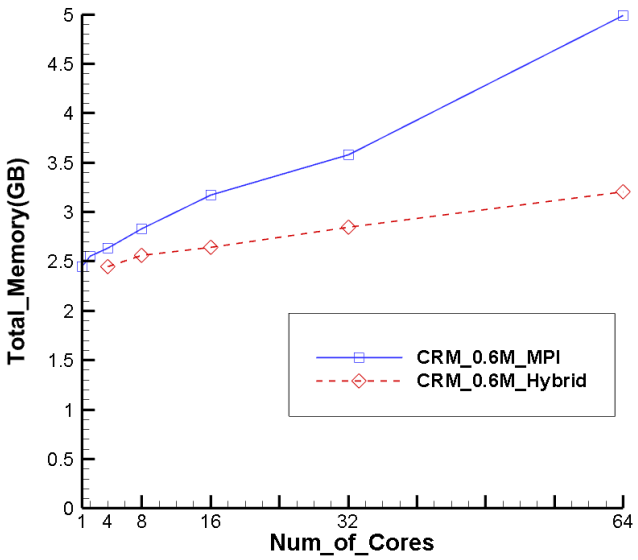


그림 4.12 노드 내에서 코어 수에 따른 필요 메모리

제 5 장 결 론

기존의 MPI 병렬기법이 적용된 혼합격자 기반 압축성 유동해석 프로그램에 OpenMP 병렬기법을 추가, 적용하여 MPI와 OpenMP를 모두 사용 가능한 Hybrid 프로그램을 만들어 병렬 최적화를 수행하였다.

유한체적법(Finite Volume Method)을 기반으로 하는 유동 해석 프로그램에서 MPI 병렬 기법만 사용할 경우, 코어를 MPI 통신 셀로 인한 메모리 부족으로 인해 대용량 계산에 제한이 있거나 MPI 통신 시간 비율이 증가하여 병렬 확장성 및 성능이 감소한다. 이때 통신량을 감소시키기 위해 OpenMP를 추가하여 Hybrid 프로그램을 작성하려면, 공유 메모리 환경에서 흔히 발생하는 오류인 경합조건에 대해 이해하고 계산 시스템에 대한 이해를 바탕으로 적절하게 쓰레드를 배치함으로써 병렬 성능 감소를 방지해야 한다.

또한 본 연구는 외재적(explicit) 시간차분법에 대해 진행하였으며 추후 내재적(implicit) 시간차분법에 대한 OpenMP 병렬화가 필요하다. 그 과정에서 OpenMP는 주로 반복문을 병렬화하는 기법이기 때문에 행렬의 병렬 계산을 위해서는 알고리즘의 전면 수정이 필요할 것으로 예상된다.

참 고 문 헌

[1] Peter Pacheco, *An Introduction to Parallel Programming.* Jpub, S.-M. KIM, 2015.

[2] Rolf Rabenseifner, Georg Hager, and Gabriele Jost, “Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes,” in 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, 2009

[3] Georg Hager, *Introduction to High Performance Computing for Scientists and Engineers.* CRC Press, 2010.

[4] J.-K. Kim, K.-J. Jang, T.Y. Kim, D.-R Cho, S.-D. Kim and J.-Y. Choi, "OPENMP PARALLEL PERFORMANCE OF A CFD CODE ON MULTI-CORE SYSTEMS," Journal of Computational Fluids Engineering, Vol. 18, No. 1, pp. 83~90. 2013.

[5] C.-W. Shu and S. Osher, “Efficient implementation of essentially nonoscillatory shock-capturing schemes,” Journal of Computational Physics, vol. 77, no. 2, pp. 439 - 471, 1988.

[6] J. S. Park, S.-H. Yoon, and C. Kim, “Multi-dimensional limiting process for hyperbolic conservation laws on unstructured grids,” Journal of Computational Physics, vol. 229, no. 3, pp. 788 - 812, 2010.

[7] J. S. Park, T. K. Chang, and C. Kim, “Higher-order multi-dimensional limiting strategy for correction procedure via reconstruction,” in 52nd Aerospace Science Meeting, pp. 2014 - 0772, 2014.

[8] S.-H. Yoon, C. Kim, and K.-H. Kim, “Multi-dimensional limiting process for three-dimensional flow physics analyses,” Journal of Computational Physics, vol. 227, no. 12, pp. 6001 - 6043, 2008

[9] Netzer R., Miller B., “Experience with techniques for refining data race detection.”, International Workshop on Languages and Compilers for Parallel Computing Lecture Notes in Computer Science, vol 757. pp 449-463

[10] https://commonresearchmodel.larc.nasa.gov/

[11] https://aiaa-dpw.larc.nasa.gov/

[12] mpi-forum.org, MPI: A Message-Passing Interface Standard, Ver. 3.0, 2012

[13] OpenMP.org, OpenMP Application Program Interface, Ver. 4.0, 2013

[14] https://www.hpc2n.umu.se/resources/hardware/kebnekaise

[15] http://cpudb.stanford.edu

Abstract

Optimization on Hybrid MPI-OpenMP Parallel Program for Compressible Flow Analysis Based on Unstructured Mixed Grid

KIM SANGHYEOK
Department of Mechanical and Aerospace Engineering
The Graduate School
Seoul National University

This paper deals with a hybrid MPI-OpenMP parallel optimization by adding OpenMP to a MPI program for compressible flow analysis based on unstructured mixed grid. Most of the flow analysis programs based on the finite volume method use MPI parallel method. However, when many cores are used, the parallel performance is decreased due to the increase of the MPI communication time ratio. To improve this, a hybrid program is created by applying the OpenMP parallel method.. In the process, Race condition that occurs when multiple threads simultaneously store values in shared memory has been found and improved, and Hybrid parallel optimization methods such as OpenMP scheduling and thread placement according to the system are described. In addition, the hybrid program tested in various grids and conditions confirmed that parallel scalability and performance were improved compared to the MPI program.

keywords : Computational Fluid Dynamics, OpenMP, MPI, Hybrid parallel programming, Speedup, Scalability
Student Number : 2017-26436