공 학 석 사 학 위 논 문

Xeon Phi와 병렬 프로그래밍을 이용한 파동 전파 모델링

2017년 2월

부경대학교 대학원

에너지자원공학과

류 동 현

공 학 석 사 학 위 논 문

Xeon Phi와 병렬 프로그래밍을 이용한 파동 전파 모델링

지도교수 하 완 수

이 논문을 공학석사 학위논문으로 제출함

2017년 2월

부경대학교 대학원

에너지자원공학과

류 동 현

류동현의 공학석사 학위논문을 인준함

2017년 2월

주 심 공학박사 최 요 순 (인)

위 원 공학박사 하 완 수 (인)

위 원 공학박사 엄 정 기 (인)

초 록

본 연구에서는 병렬 연산 가속기인 Xeon Phi 와 병렬 프로그래밍을 통해 효율적인 시간영역 파동 전파 모델링을 제시하였다. 본 연구에서 적용한 병렬 프로그래밍과 Xeon Phi 의 매니코어 특징을 이용함으로써 기존 CPU 환경보다 효율적인 모델링을 수행할 수 있다. 이를 확인하기 위해 송신원 개수와 격자 크기를 바꿔가며 실험하였고 속도 모델 적용을 통해 컴퓨터 계산 시간 및 정확도를 비교하였다. 모델링을 기초로 차후 파형역산, 구조 보정 등 고성능 자료처리에도 적용하여 성능 향상을 기대해 볼 수 있다.

주요어 : 제온 파이(XeonPhi), 가속기, 파동 전파 모델링, 병렬 연산, 유한차분법
학 번 : 201555173

목 차

List of Figures

List of Tables

1. 서 론

석유·가스 개발을 위한 물리탐사 분야에서 탄성파 탐사는 가장 기본적으로 이용하는 기술이다. 지하구조에 따른 탄성파 거동을 기록·분석하여 이를 정확히 모사할수록 고해상도의 지하구조 영상화가 가능하다. 현재까지 고해상도에 따른 자료의 양과 처리량은 계속해서 증가되고 있는 상황이고 이에 따라 고성능 자료 처리 기술의 개발이 활발히 이루어지고 있다.

고성능 자료처리 기술은 고성능 컴퓨터 및 가속기를 사용하거나, 병렬 처리 소프트웨어를 이용하는 것으로 자료 처리 시간 및 비용과 같은 계산 자원을 줄일 수 있다. 최근에는 GPU(그래픽 프로세서 유닛) 나 Xeon Phi Coprocessor(제온 파이 보조 프로세서) 등 연산 가속기 활용을 많이 하고 있는 추세이다(Suh and Wang, 2011; Banas and Kruzel, 2014; Mori et al., 2014; Kim et al., 2013; Jeffer et al., 2013; Reinder et al., 2014). 주요 프로세서 제조사 중 하나인 Intel(인텔)에서 출시한 Xeon Phi Coprocessor 는 기존 CPU 에 비해 코어의 개수가 많고, 확장된 벡터 레지스터 등 병렬프로그래밍 및 연산에 적합한 도구이다(Chrysos et al., 2012).

본 연구에서는 Xeon Phi Coprocessor 와 병렬 프로그래밍을 이용하여 시간영역 파동 전파 모델링을 구현하였다. 효율적인 파동 전파 모델링을 위해서는 해당 프로세서의 구조적인 특징과 프로그래밍에 대해 구체적으로 이해 할 필요가 있다. 또한 Xeon Phi 로 구현이 가능한 병렬화 모델을 파악하여 연구에 적용하여야 한다. 이를 위해 Xeon Phi 의 구조에 대해 분석해보고 관련된 병렬 프로그래밍 기술에 대해 살펴보고자 한다. 다음으로 기본적인 파동 전파 모델링 알고리즘에 적용하여 병렬 모델링 알고리즘을 개발한 후 CPU 와 Xeon Phi 에서 비교를 통해 효율을 알아 볼 것이다.

모델링의 방법은 많은 유한차분법 중 매질의 물성과 파동장을 격자에 정의하는 격자 기반의 유한차분법을 적용하였다(Kelly et al., 1976; Alterman et al., 1968; Alford et al., 1974). 여러 가지 실험을 통해 CPU와 Xeon Phi의 연산 시간을 비교 해보고, 속도 모델에 적용해 봄으로써 정확도 및 연산 효율을 살펴볼 것이다. 파동 전파 모델링은 고해상도 자료 처리의 기초가 되는 기술로써 Xeon Phi 를 이용한다면 다른 자료 처리 연구에도 효율적으로 활용될 수 있을 것이다.

2. 관련 연구

2.1 Xeon Phi Coprocessor(제온 파이 보조프로세서)

인텔에서 출시한 Xeon Phi 는 MIC(Many Intergrated Core) 구조로서 매니코어 코프로세서(Manycore Coprocessor)이다(Fig. 2.1). 코프로세서란 주 프로세서(CPU)에서 특정한 연산 작업을 오프로딩 하여 그 작업을 빠르게 수행 할 수 있도록 디자인 된 마이크로 프로세서 카드(chip)를 말한다. 현재 까지 출시된 Xeon Phi는 최대 61개 이상의 코어로 이루어져 있으며 각 코어는 양방향성 링 버스로 연결되어 있다. 벡터 연산 유닛(VPU-Vector Processing Unit)은 512 bit 폭을 가지고 있고 SIMD(Single Instruction Multiple Data) 엔진을 통해 한 번의 연산으로 16개의 일배수 배정밀도(64bit) 연산이나 8개의 이배수 배정밀도의 소수점 연산의 수행을 지원한다. L1 cache(캐시)는 데이터 cache와 명령어 cache 2개로 구성되어 있으며 L2 cache 는 512KB 의 용량을 갖는다.

또한 멀티 스레딩으로 (SMT-Simultaneous Multi-Threading) 동시에 4개의 스레드를 수행 할 수 있고, 각 코어마다 4개의 하드웨어 스레드를 제공해 멀티 스레딩과 연결하여 “4*코어수“ 만큼 스레드를 사용 할 수 있다. Fig. 2.2 처럼 코어의 스레드 분배도 설정이 가능해 연구 환경에 맞게 유연하게 사용하여 최대 성능을 이끌어 낼 수 있다.

이렇게 Xeon Phi는 구조적인 것뿐만 아니라 다양한 실행 옵션을 제공해 성능 향상을 시키고자 하는 프로그램의 특성에 맞게 적용할 수 있다는 장점을 가지고 있다. Fig. 2.3 에서 다양한 실행 옵션에 따른 설명이 나와 있다(Jeffers et al., 2014).

Fig. 2.1 Intel(R) Xeon Phi Coprocessor (7120P)

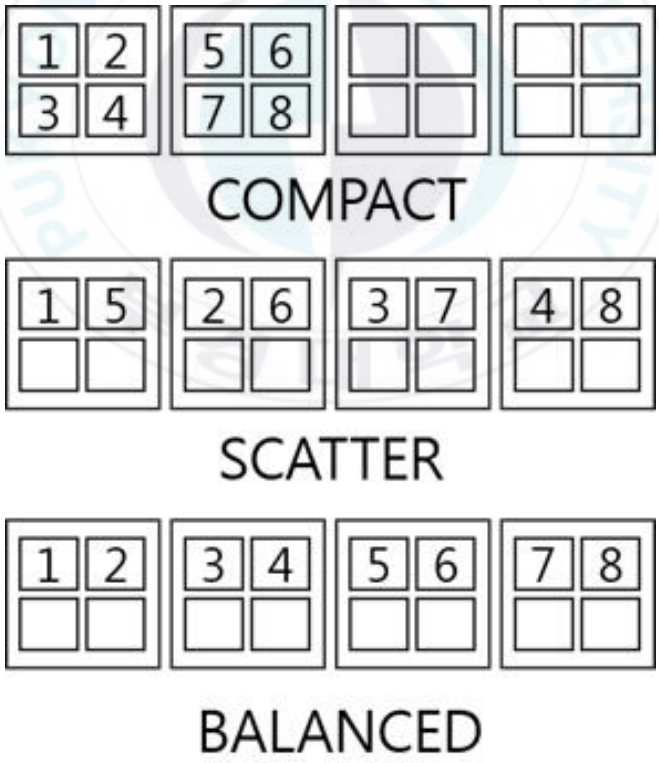


Fig. 2.2 Thread distribution algorithm of Xeon Phi execution.

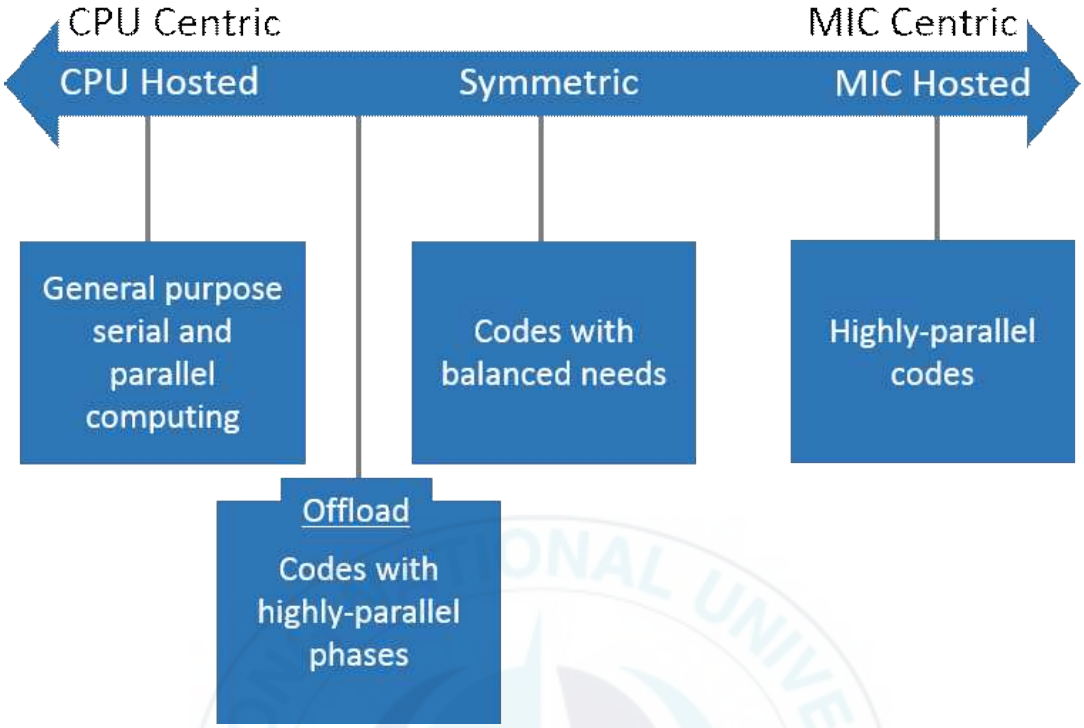


Fig. 2.3 Various Execution options of Xeon Phi

2.2 OpenMP

멀티코어를 가지는 CPU 혹은 Xeon Phi의 장점을 살리기 위해서는 이에 맞는 적절한 프로그래밍이 필요하다. 순차적 프로그래밍이 된 코드는 멀티코어에서 사용하더라도 병렬성을 인식하지 못하기 때문에 성능 향상은 이루어 질 수 없다. 따라서 순차적 프로그래밍을 병렬적 프로그래밍으로 코드를 바꾸거나 혹은 병렬적으로 새로 작성하는 방법이 있다. 하지만 기존에 있는 순차 프로그래밍을 병렬 프로그래밍으로 수정 하는 것은 많은 시간을 필요로 하며 새로운 병렬 프로그래밍 코드를 작성하는 것은 알고리즘 복잡성의 증가로 매우 어렵다. 이러한 대체방안으로 OpenMP 가 나왔다. OpenMP는 다양한 플랫폼에서 프로세스간의 메모리 공유를 하여 병렬 계산을 도와주는 API(Application Programming Interface)의 집합체이다. 동시 멀티 프로세싱(SMP -Symmetric Mutiprocessing)과 리눅스 체제의 언어, 컴파일러(Compiler) 대부분을 지원한다. OpenMP는 병렬화를 목표한 부분에 OpenMP 지시어(#pragma omp parallel)를 작성하여 사용할 수 있으며, 컴파일러는 OpenMP 지시어를 인식하여 자체적으로 병렬화가 가능한 작업은 다수의 스레드로 병렬 처리를 한다(Fig. 2.4)(Babara et al., 2007).

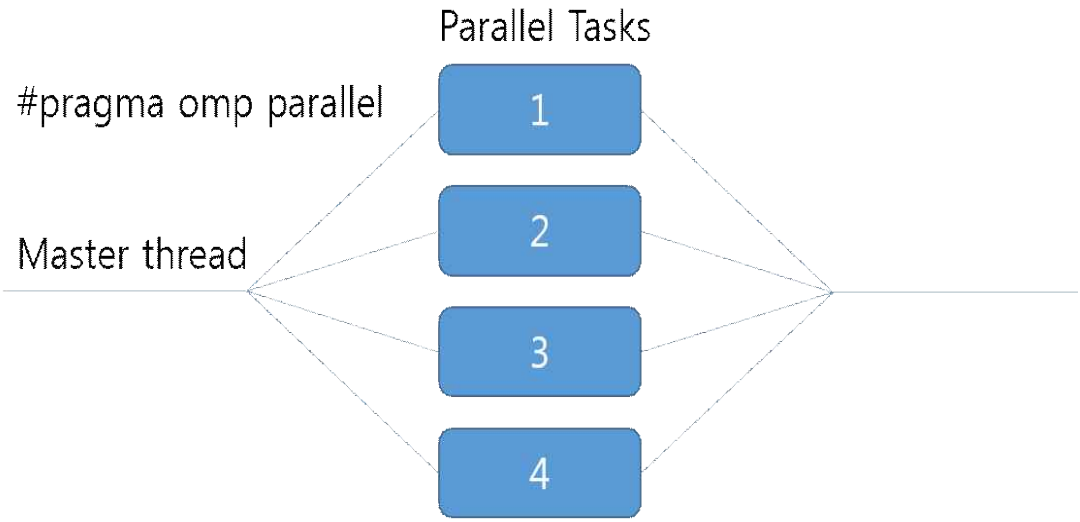


Fig. 2.4 OpenMP parallel structure

2.3 MPI(Message Passing Interface)

다른 병렬 프로그래밍의 기술로 MPI(Message Passing Interface)가 있다. MPI 는 프로세서들 사이의 통신을 위해 코드에서 호출해 사용하는 서브루틴(Fortran) 또는 함수(C)들의 라이브러리이다. 또한 MPI는 프로세서 간에 메모리를 공유하지 않는 분산 메모리 시스템에서 병렬로 연산하며 정보를 교환하기 위한 규약으로 OpenMP와 같이 동일한 메모리 공간에 접근할 수 있는 공유 메모리 시스템과는 차이점이 있다. 분산메모리 구조는 2개 이상의 프로세서가 네트워크로 연결 되어 있는 시스템을 의미한다. 각 프로세서는 코어에 대응해서 독립적으로 계산을 수행하기도 하고 네트워크를 통해 데이터를 주고받거나 마스터 프로세서로 모으는 등의 작업을 수행할 경우에 MPI를 사용하게 된다.(Fig. 2.5) 이와 같은 병렬 방법은 매니코어를 가지는 Xeon Phi나 멀티코어 CPU에 많이 적용되며 계산에 참여하는 프로세서의 개수가 많을수록 성능과 생산성이 증가한다. 데이터 송수신 방법에는 크게 점대점 통신과 집합 통신이 있고 MPI_SEND, MPI_RECV, MPI_BCAST, MPI_REDUCE 와 같은 여러 루틴(routine)을 사용해 통신을 수행할 수 있다(Pacheco, 2011).

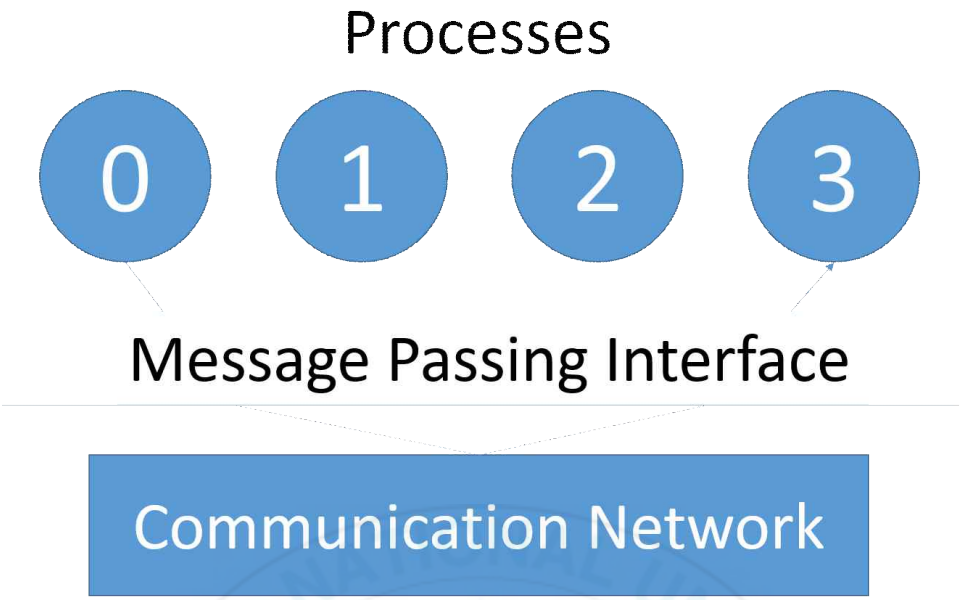


Fig. 2.5 MPI message passing

3. 이 론

3.1 파동전파모델링 알고리즘

3.1.1 시간-공간 영역 유한차분법 모델링

본 연구에서는 유한 차분법 중 격자 기반의 유한 차분법 이용하여 모델링을 수행하였다(Kelly et al., 1976).

먼저 등방성 음향 파동 방정식은 다음과 같다.

$$\frac{\partial^2 p}{\partial t^2} = v^2 \nabla^2 p \tag{1}$$

여기서 t 는 시간, p 는 시간에서의 압력장, v 는 P파 전파 속도를 나타낸다. 이것을 3차원 공간 영역을 이용해 전개하면 다음과 같다.

$$\frac{\partial^2 p}{\partial t^2} = v^2 \left(\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2} + f \right) \tag{2}$$

또한 시간에 대한 2차미분항을 유한 차분법으로 전개하여 근사 시키면 다음과 같다.

$$\frac{p^{t+\triangle t} - 2p^t + p^{t-\triangle t}}{\triangle t^2} = v^2 \left(FD_x + FD_y + FD_z + f \right) \tag{3}$$

여기서 t 는 시간 축으로 p^t 는 $p(x,y,z,t)$ 이며 $\triangle t$ 는 시간격자를 의미한다. 우변 또한 각 공간에 대한 2차미분항을 유한차분법으로 전개하여 근사 시키면 다음과 같이 표현된다.

$$FD_x = \frac{1}{\triangle x^2}\left(p^t_{x-\triangle x,y,z} - 2p^t_{x,y,z} + p^t_{x+\triangle x,y,z}\right) \quad (4)$$

$$FD_y = \frac{1}{\triangle y^2}\left(p^t_{x,y-\triangle y,z} - 2p^t_{x,y,z} + p^t_{x,y+\triangle y,z}\right) \quad (5)$$

$$FD_z = \frac{1}{\triangle z^2}\left(p^t_{x,y,z-\triangle z} - 2p^t_{x,y,z} + p^t_{x,y,z+\triangle z}\right) \quad (6)$$

$p^t_{x,y,z}$ 에서 x, y, z 는 공간 축이며 $\triangle x$, $\triangle y$, $\triangle z$ 는 공간격자를 의미한다.

만약, 공간에 대해 n차미분항을 유한차분법으로 전개해 나타내면 다음과 같다.

$$FD_x = \frac{1}{\triangle x^2}\left(c_0 p^t_{x,y,z} + \sum_{n=1}^{n/2} c_n\left(p^t_{x+n\triangle x,y,z} + p^t_{x-n\triangle x,y,z}\right)\right) \quad (7)$$

$$FD_y = \frac{1}{\triangle y^2}\left(c_0 p^t_{x,y,z} + \sum_{n=1}^{n/2} c_n\left(p^t_{x,y+n\triangle y,z} + p^t_{x,y-n\triangle y,z}\right)\right) \quad (8)$$

$$FD_z = \frac{1}{\triangle z^2}\left(c_0 p^t_{x,y,z} + \sum_{n=1}^{n/2} c_n\left(p^t_{x,y,z+n\triangle z} + p^t_{x,y,z-n\triangle z}\right)\right) \quad (9)$$

식 (7) ~ (9) 의 c_0 부터 $c_{n/2}$ 는 유한 차분 계수이다.

따라서 2차미분항을 기준으로 시간 순서로 볼 대 과거의 파동장 $p^{t-\triangle t}$ 과 현재의 파동장 p^t 를 알고 있다면 다음과 같이 미래의 파동장 $p^{t+\triangle t}$ 을

나타 낼 수 있다.

$$p_{x,y,z}^{t+\triangle t} = 2p_{x,y,z}^{t} - p_{x,y,z}^{t} + \triangle t^2 v^2 (FD_x + FD_y + FD_z + f_{(D,t)}) \qquad (10)$$

여기서 $f_{(D,t)}$ 는 송신원 파형 요소, $p_{x,y,z}^{t}$ 는 파동장, D 는 송신원 위치를 나타낸다. 공간의 각 점에서 새로운 파동장은 현재($p_{x,y,z}^{t}$)와 이전($p_{x,y,z}^{t-\triangle t}$) 값을 이용해 계산한다. 이러한 특성을 이용해 모든 격자점을 업데이트 수 있다.

3.1.2 경계 조건

파동 전파 혹은 방정식 모델링에서 중요한 문제 중 하나는 모델의 외곽에서 발생하는 인공적인 반사파를 제거 하는 것이다. 실제 지하매질은 반무한 매질이므로 이러한 반사파가 발생하지 않지만, 수치해석을 통한 모델링의 경우 유한한 크기의 모델을 설정하여 수행하게 되므로 모델의 외곽경계에서 인공적인 반사파가 발생한다. 이러한 반사파는 탄성파 모델링 결과의 해석을 어렵게 하는 요소로서 흡수 경계 조건을 적용하여 인공적인 반사파를 제거하는 것이 바람직하다.

본 연구에서는 다양한 흡수 경계 조건 중 Reynolds 경계 조건을 이용하였다. Reynolds 경계 조건은 음향 파동 방정식에서 일방향 파동 방정식으로 분리한 후, 각 경계조건에 바깥쪽으로 파가 전파하도록 적용해 주는 것이다. 인수분해를 통해 다음과 같이 일방향 파동 방정식으로 분리할 수 있다.(Reynolds et al., 1978)

$$\left(\frac{1}{v}\frac{\partial}{\partial t}+\frac{\partial}{\partial x}\right)\left(\frac{1}{v}\frac{\partial}{\partial t}-\frac{\partial}{\partial x}\right)=0 \tag{1}$$

양의 방향과 음의 방향으로 분리 된 파동 방정식을 시간과 공간격자를 통해 나타낼 수 있다.

$$p_x^{t+\triangle t}=\frac{c\triangle t}{\triangle x}(p_{x-\triangle x}^t-p_x^t)+p_x^t \tag{2}$$

$$p_x^{t+\triangle t}=\frac{c\triangle t}{\triangle x}(p_x^t-p_{x-\triangle x}^t)+p_x^t \tag{3}$$

1차원에서 각각 양의 방향과 음의 방향으로 전파하는 파에 대해 적용하면

Reynolds 경계 조건을 만족 시킬 수 있다.

이렇게 격자 기반의 유한 차분법 및 경계조건을 적용한 파동 전파 모델링의 알고리즘은 Table. 3.1 과 같다.

Table. 3.1 A time-domain wave propagation modeling algorithm. The initializing process includes allocating arrays and loading velocity and source information.

```
Initialize
FOR each time Do
    Solve finite-difference equation
    Inject source
    Apply boundary condition
    Time march
    Write wavefield
END FOR
```

3.2 MPI 를 이용한 병렬 모델링

본 연구에서 사용하는 MPI 병렬 모델링 알고리즘은 다중 송신원을 사용하게 만든 알고리즘이다. 사용하는 송신원 개수가 많을수록 보다 정확한 이미지와 정보를 얻을 수 있다는 사실은 많이 알려져 있다. 따라서 하나의 프로그램이 하나의 프로세스(코어)에서 개별적으로 실행하도록 설계하여 기본 틀은 순차적인 파동전파모델링을 따르고 필요한 정보만 MPI 통신을 이용, 프로세서에서 개별적으로 연산을 수행하게끔 하였다.

MPI를 이용하여 파동 전파 모델링을 위한 유사 알고리즘을 Table 3.2에서 제시하였다. MPI의 집합 통신(MPI_BCAST)를 이용해 속도와 소스 정보를 각 프로세서에 할당해주고 한 개의 프로세서는 한 개의 송신원(파동장)을 계산하게 된다(Fig 3.1).

Table. 3.2 A time-domain wave propagation modeling algorithm for Multiple source position with MPI. The initializing process includes allocating arrays and loading velocity and source information.

```
Initialize
MPI_BCAST :: velocity, source information
FOR 1 shot per each processor(core) using MPI function
FOR each time Do
    Solve finite-difference equation
    Inject source
    Apply boundary condition
    Time march
    Write wavefield
END FOR
END FOR
```

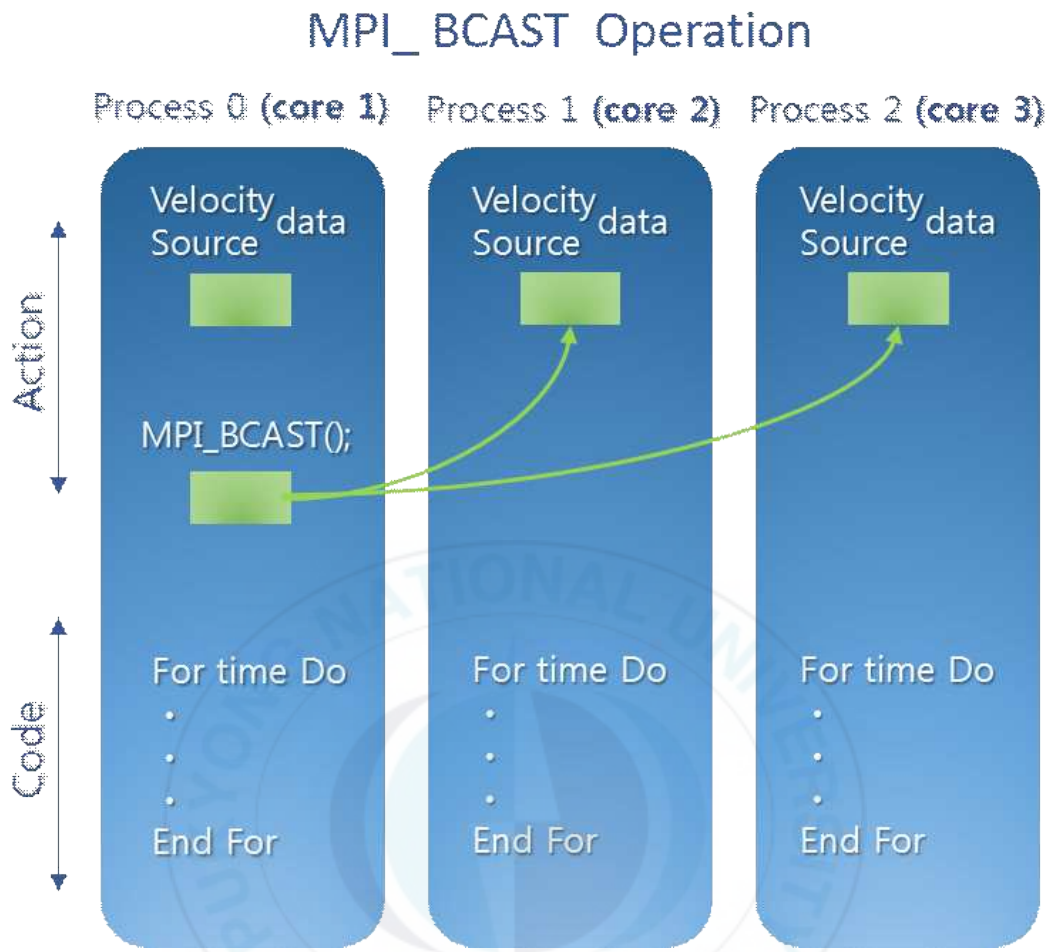


Fig. 3.1. MPI_BCAST Operation for Multiple source position. During a broadcast, one process sends the data(velocity, source) to all processes in a communicator.

3.3. OpenMP 를 이용한 병렬 모델링

OpenMP를 이용한 병렬 모델링에서는 기존에 최적화 및 OpenMP를 이용하여 구현한 유한차분법 알고리즘(Cédric, 2014)에서 송신원 추가와 경계조건 적용을 통해 파동전파모델링을 구현하였다. 해당 알고리즘은 Table 3.3, Fig 3.2 과 같다.

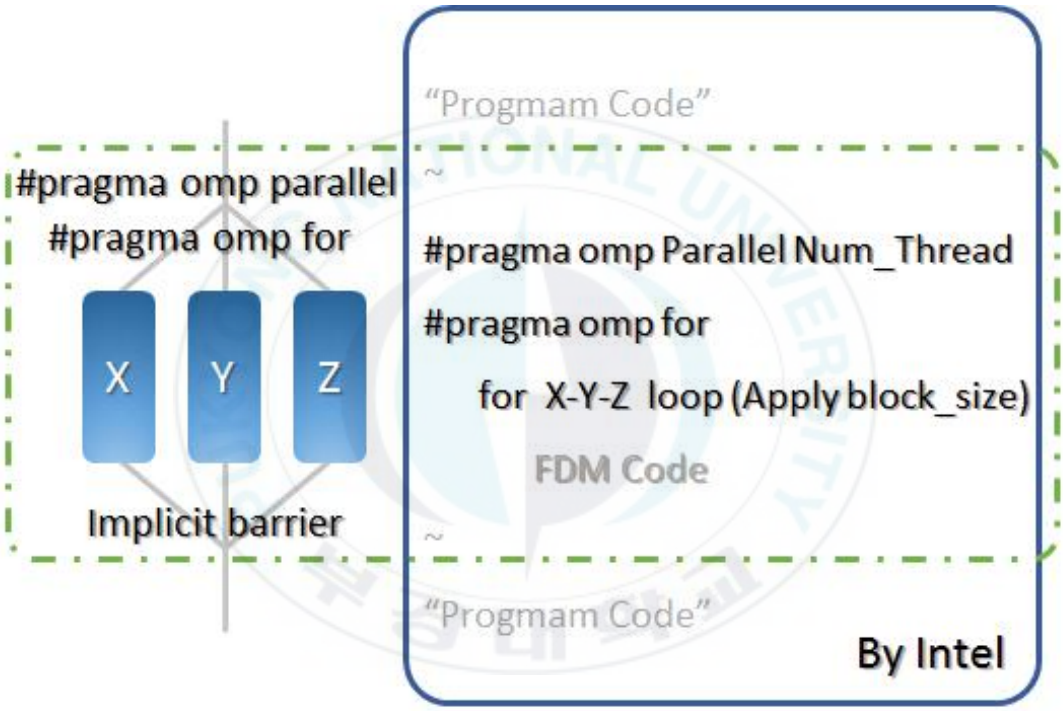


Fig. 3.2. Optimized finite-difference equation using OpenMP directive(#pragma omp parallel, #pragma omp for) by intel.

Table. 3.3 A Time-domain wave propagation modeling algorithm for OpenMP Parallel programming. The initializing process includes allocating arrays and loading velocity and source information.

```
Initialize
FOR each time Do
    Solve Optimized finite-difference equation using OpenMP
    Inject source
    Apply boundary condition
    Time march
    Write wavefield
END FOR
```

경계조건 적용 시 순차적인 경계조건은 이미 최적화 및 병렬화 된 유한차분법 연산 알고리즘에 영향을 주기 때문에 연산성능이 떨어지게 된다. 따라서 경계조건 또한 최적화와 OpenMP를 이용한 경계조건으로 바꿔 줄 필요가 있다.

Table. 3.4 는 최적화 위해 변경된 경계 조건 연산 알고리즘을 나타낸 것이다.

Table. 3.4 3D optimized boundary condition using OpenMP and cache block.

```
#pragma omp parallel num_thread
#pragma omp for collapse
FOR each Z_Y_X Do (Apply block_size)
    right boundary condition
    left boundary condition
    front boundary condition
    back boundary condition
    bottom boundary condition
END FOR
```

OpenMP 의 #pragma omp for 는 일반적으로 지시어 영역 안에 for 루프가 2개 이상 중첩되면 가장 외곽에 있는 for 루프 하나에 대해서만 병렬화를 지원한다. 3차원에 대해 경계조건을 추가하는 것을 생각한다면 collapse 보조 지시어를 사용하여 내측의 루프에서도 OpenMP 병렬처리를 하게 한다. 이렇게 하면 스레드는 1개의 커다란 루프처럼 인식해 작업을 분할하는 스케쥴링을 하게 되고 병렬 효율성이 증가하게 된다. 또한 최적화에 있어 캐시 블록은 각 배열에 적절한 블록값을 지정해 값을 메모리로 가져오는데 필요한 시간을 줄여 성능을 올리게 한다.

OpenMP와 캐시 블록을 통한 최적화된 경계조건의 알고리즘을 사용한다면 Xeon Phi의 성능은 좋아질 뿐만 아니라 CPU에서도 나은 성능을 보일 것이다.

4. 실험 구성 및 검증

본 연구에서는 MPI 및 OpenMP를 적용한 병렬 파동전파모델링 알고리즘을 제시하였으며, 수치 예제를 통해 Xeon Phi 와 CPU 의 연산 시간 및 효율을 살펴보고자 하였다.

기본적으로 MPI는 2차원 모델을 적용하여 송신원 개수에 따라 효율을 비교하였고 OpenMP는 3차원 모델을 적용하여 모델 크기에 따라 효율을 비교하였다. 또한 각각 실제 속도모델을 적용해 성능 차이를 알아보았다. 연산 비교에 있어 계산강도(AI, Arithmetic Intensity - 메모리 전송 바이트당 부동 소수점 작동 횟수(Flops))를 제외하고 연산시간을 이용하였다. 연산시간은 실행시간을 바탕으로 수행하고 실행횟수의 평균으로 나타낸다.

Table 4.1 은 본 연구의 실험 환경으로써 CPU 모델링은 Xeon E5-2650 v2 의 단일 CPU(8코어), Xeon Phi 모델링은 단일 카드(61코어)를 사용하여 계산하였다. 컴파일러는 모두 Intel compiler 15.0.2 버전을 사용했고 모든 실험은 정확한 비교를 위해 Xeon Phi 는 직접실행(Native) 방식을 채택하여 컴파일은 CPU 에서 수행하고 실행은 Xeon Phi 에서 하도록 하였다.

Table 4.1 Environments of CPU and XeonPhi Coprocessor

	CPU 프로세서	Xeon Phi 보조 프로세서
모델명	Intel Xeon E5-2650 v2	Intel Xeon Phi 7120P
Core 수	소켓당 8개 (스레드8개)	카드당 61개 (스레드 244개)
Core clock	2.60GHz	1.238GHz
Memory size	256GB	16GB

4.1 알고리즘 정확도 검증

각 수치 예제를 적용하기에 앞서 MPI 병렬 알고리즘에서는 격자연산에 있어 순차적 알고리즘과 같기 때문에 단일 모델링 시, 정확도 검증이 필요 없으나 OpenMP 를 이용한 3차원 병렬 알고리즘은 멀티스레드 사용으로 정확도 검증이 필요하다. 이를 위해 CPU 와 Xeon Phi 에서 모델링을 수행한 후 그 결과를 비교하였다.

먼저 3차원 모델링에 있어 1.5km/s 의 상 속도를 갖는 층을 대상으로 동일한 지점에 음원과 수진기를 위치시켜 신호의 정확성을 비교하였다.

모델링 환경은 Table 4.2 과 같다. 스레드 분배 방식은 Scatter를 이용하였으며 스레드 블록은 124, 1, 2 로 지정하였고 격자간격은 X,Y,Z 축 모두 동일하며, 샘플링간격은 2ms로 총 4초간 기록하였다. 시간에 때해 2차, 공간에 대해 8차 유한 차분법을 사용하였다. 순수 모델링 연산 비교를 위해 경계조건은 고정단 경계조건(Kreyszig, 2010)을 사용하였다.

Table. 4.2 3D modeling parameter about algorithm accuracy

parameter	value
Number of X-Y-Z axis	208,208,208
Number of X-Y-Z Thread block	124,1,2
Grid interval(m)	20
Sampling interval(ms)	2
frequency(Hz)	10

Fig. 4.1 는 CPU와 Xeon Phi 에서 모델링 결과이다. Fig 4.1 의 (C)에서 파형 및 진폭이 일치함을 확인할 수 있었으며, 이를 통해 알고리즘이 CPU와 Xeon Phi 에서 연산이 정확히 이루어짐을 확인하였다.

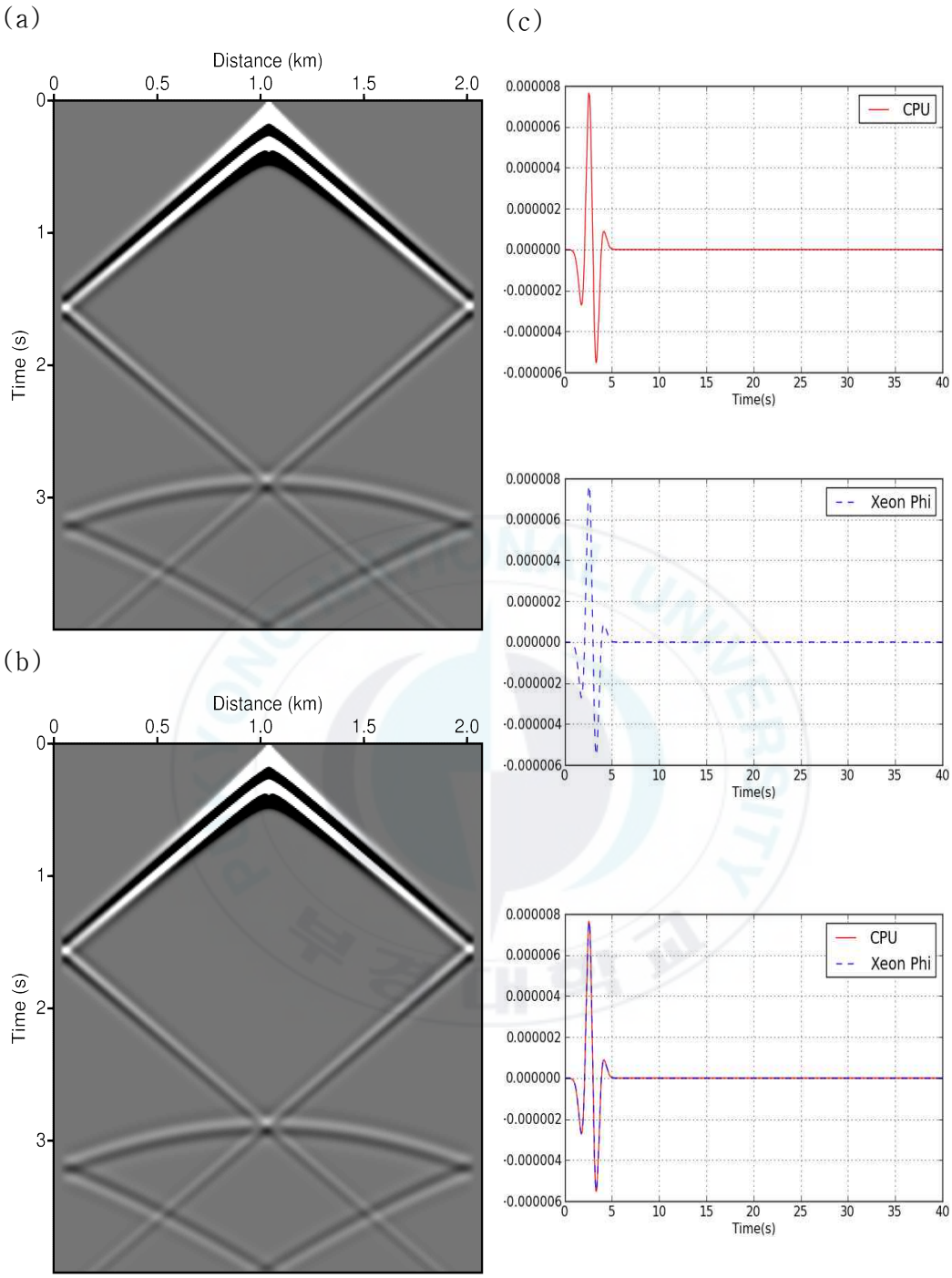


Fig. 4.1 Surface seismogram (x=1km) from CPU(a) and Xeon Phi programs(b). (c) Traces extracted from the source. CPU and Xeon Phi curves show the traces from the CPU and Xeon Phi program.

4.2 2차원 수치 예제

4.2.1 송신원 개수에 따른 비교 및 속도 모델 적용

파동 전파 모델링에 있어 송신원 개수가 많을수록 보다 정확한 이미지와 정보를 얻을 수 있다. 따라서 MPI를 이용하여 송신원에 따라 프로세스를 할당하도록 프로그램을 구현하고 CPU와 Xeon Phi 의 연산 시간을 비교하였다. 또한 경계 조건 추가로 속도 모델에 적용해 나타난 결과를 알아보았다.

프로그램은 프로세스간의 통신은 없고 단순히 계산 시간만 비교하도록 하였다. 계산시간은 세 번 실행한 결과의 평균값을 이용하였고, 단일 CPU (8코어), Xeon Phi 단일 카드(61코어)를 사용하였다.

실험에는 3km/s 의 상 속도를 갖는 2차원 모델을 사용하였고 모델링 환경은 Table 4.3 과 같다. 샘플링간격은 2ms 로 총 8초간 기록하였다. 시간에 때해 2차, 공간에 대해 8차 유한 차분법을 사용하였고 송신원 개수를 30개부터 330개까지 변화 시켜가며 실험을 진행하였다.

Table. 4.3 2D modeling parameter about comparison according to number of shot

parameter	value
Number of X-Y axis	1000,500
Grid interval(m)	20
Sampling interval(ms)	2
frequency(Hz)	10

실험결과, 상대적으로 성능은 낮지만 매니코어를 가진 Xeon Phi 가 CPU 보다 연산시간이 더 빠르게 나왔다.(Fig. 4.2) 또한 송신원 개수가 증가 할수록 성능 차이가 커짐을 보였고 송신원 개수가 200개 이상일 때, CPU

대비 Xeon Phi 의 성능은 1.8배 가량 속도 향상을 볼 수 있었다.(Fig. 4.3)

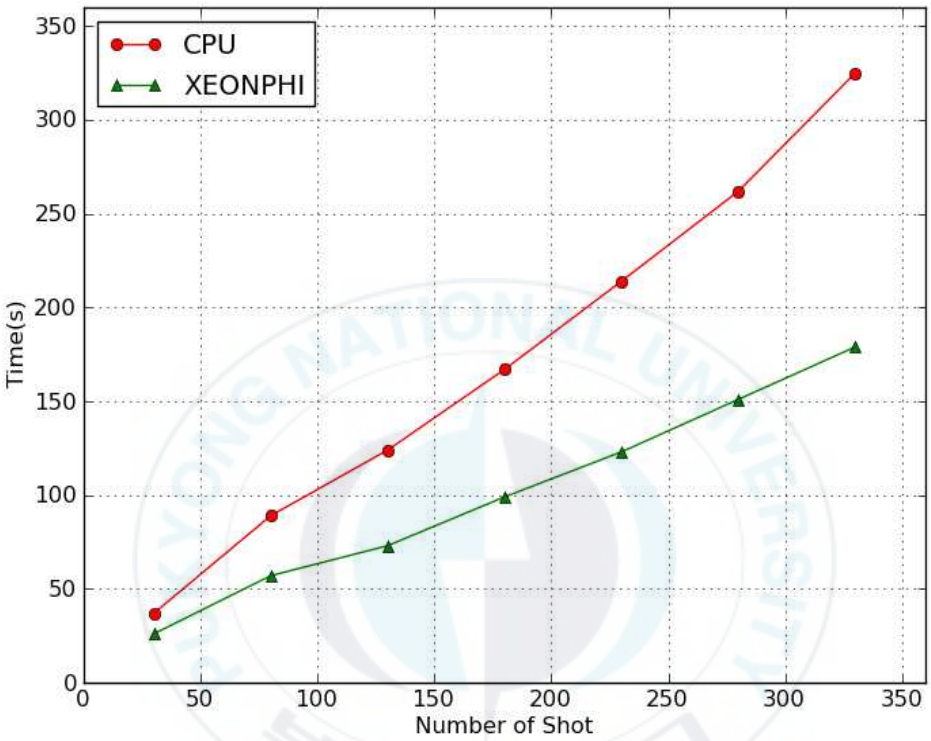


Fig. 4.2. Execution times of CPU and Xeon Phi modeling according to number of shot.

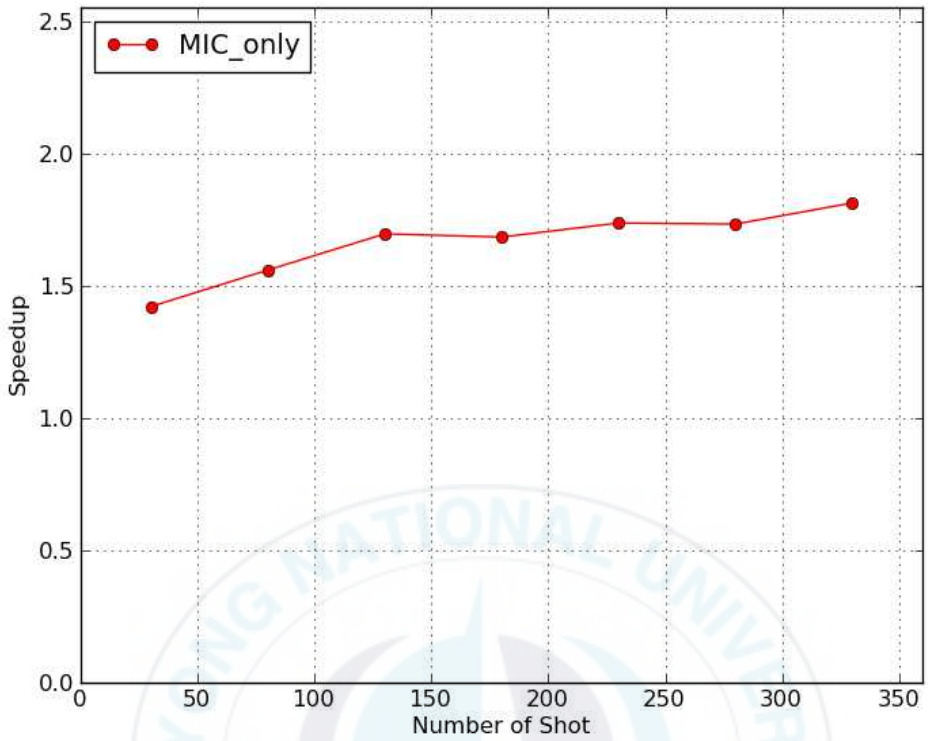


Fig. 4.3. Relative speedups of the Xeon Phi program with respect to the CPU programs using the number of same shot. No communication is included in this example.

다음으로 상속도 모델이 아닌 Marmousi 속도 모델(Versteeg, 1994)(Fig. 4.4)을 통해 모델링을 수행하였다. Marmousi 모델은 속도 모델링이나 중합 전 심도 구조보정 등의 탄성파 자료처리 기술에 대한 평가 모델로써 널리 사용되고 있다.

먼저 모델링의 환경은 Table 4.4 과 같고 총 220 개 송신원을 이용해 1ms 샘플링 간격으로 총 4초간 모델링을 수행하였다. 마찬가지로 시간에 대해 2차, 공간에 대해 8차 유한 차분법을 사용하였고 경계조건은 흡수 경계 조건(Reynolds et al., 1978)을 적용하였다.

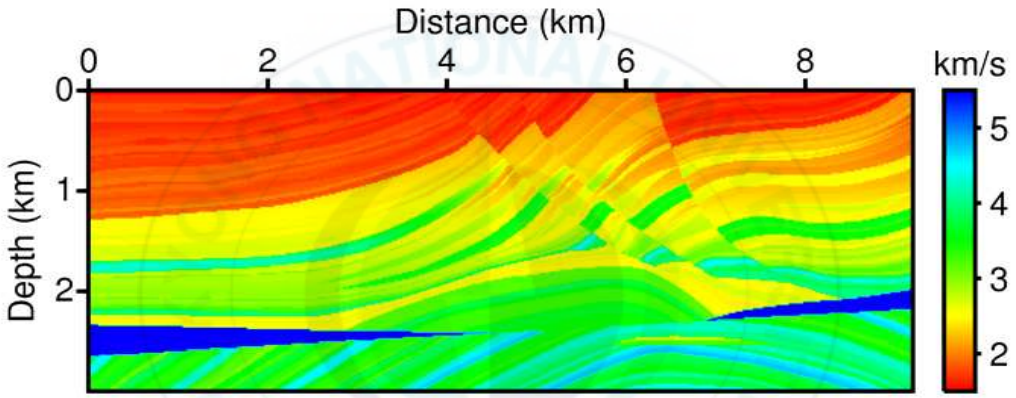


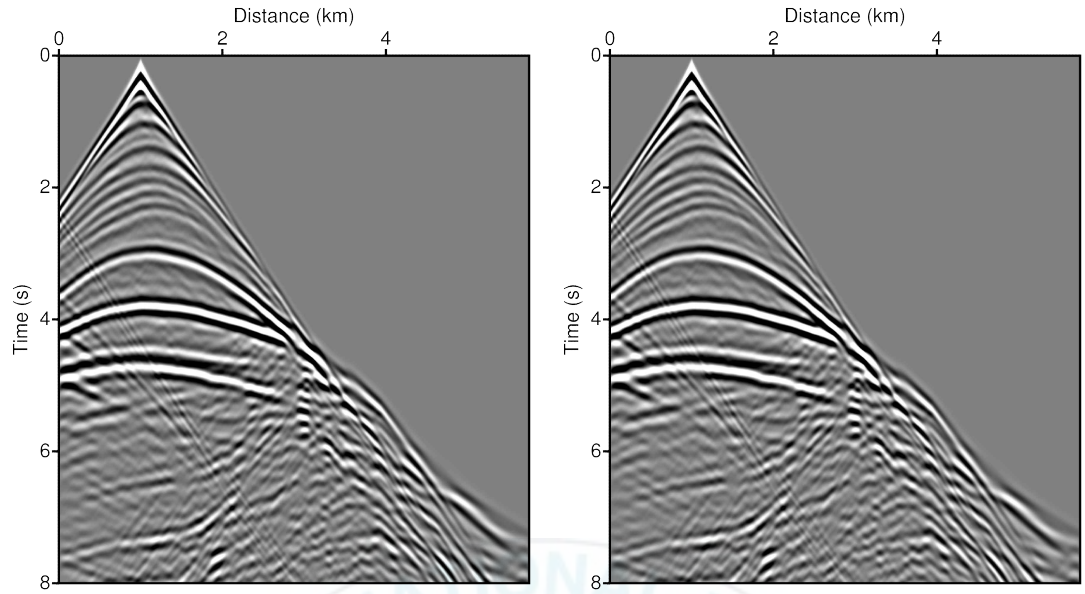
Fig. 4.4. Marmousi velocity model (Versteeg, 1994)

Table. 4.4 Apply 2D marmousi velocity modeling parameter

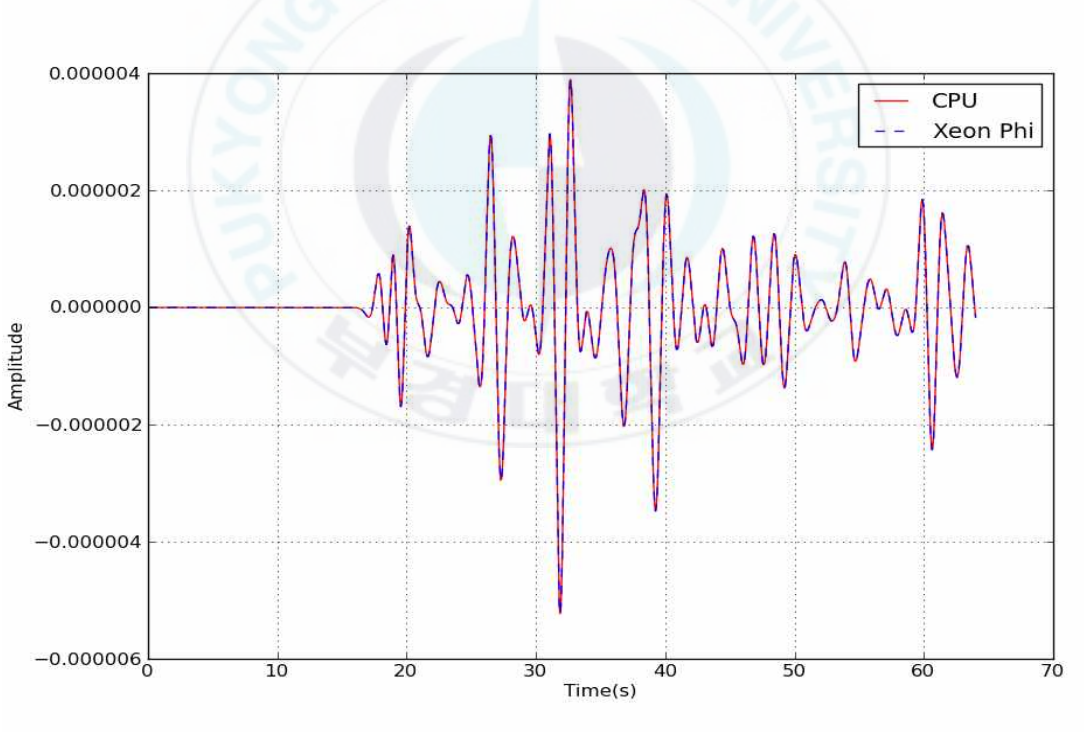
parameter	value
Number of X-Y axis	576,188
Grid interval(m)	16
Sampling interval(ms)	1
frequency(Hz)	10

실험 결과, CPU의 계산시간은 36.5초, Xeon Phi 의 계산시간은 29.1초로 Xeon Phi 를 사용하여 CPU 대비 79.8% 수준으로 계산시간이 단축됨을 알 수 있었다.

Fig 4.5 는 CPU와 Xeon Phi 에서 각각 모델링을 통해 얻은 파동장과 트레이스를 나타낸 것이다. 트레이스는 송신원에서 2km 떨어진 지점에서 추출하여 비교한 것으로, 결과를 보면 파형 및 진폭이 일치함을 나타내 정확도 또한 검증이 되었음을 보여준다.



(a) (b)



(c)

Fig. 4.5 Surface seismogram (x=1km) from (a) CPU and (b) Xeon Phi programs about 2D modeling. (c) 'CPU' and 'Xeon Phi' trace curves show the traces from the CPU and Xeon Phi programs

4.3 3차원 수치 예제

4.3.1 격자 크기에 따른 비교 및 속도 모델 적용

다음으로 OpenMP 를 이용한 3차원 알고리즘을 적용한 모델을 가지고 격자 크기를 208×208×208, 400×400×400, 608×608×608 , 800×800×800 으로 변화 시켜가며 CPU 와 Xeon Phi 의 연산시간을 비교하였다. 1.5km/s 상속도 모델로 추가 변수는 Table 4.5 와 같다. 고정단 경계조건을 사용하였고 계산 시간은 세 번 실행한 결과의 평균값을 이용하였다.

Table. 4.5 3D modeling parameter about comparison according to grid size

parameter	value
Number of X axis(equal to Y-Z)	208,400,608,800
Number of X-Y-Z Thread block	124,1,2
Grid interval(m)	20
Sampling interval(ms)	2
frequency(Hz)	10

실험 결과, Xeon Phi 의 모델링 수행시간이 CPU 보다 더 빠르게 나왔다.(Fig. 4.6) 일반적으로 모델 크기가 증가함에 따라 유한 차분 계산량도 함께 증가 하게 된다. 그에 따른 총 연산 시간 또한 비례해서 증가함을 알 수 있다. 모델 크기가 증가 할수록 성능 차이가 커짐을 보였고, 수치적으로 CPU 대비 Xeon Phi 의 성능은 1.5 ~ 2배 가량 속도 향상을 볼 수 있었다.(Fig. 4.7)

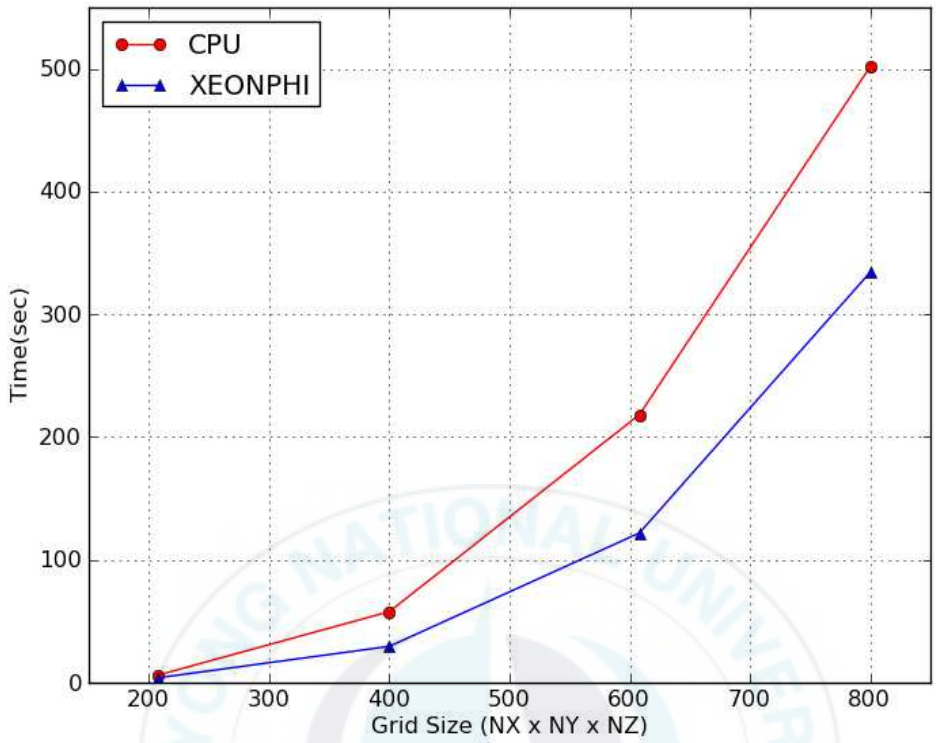


Fig. 4.6. Execution times of CPU and Xeon Phi modeling according to model size.

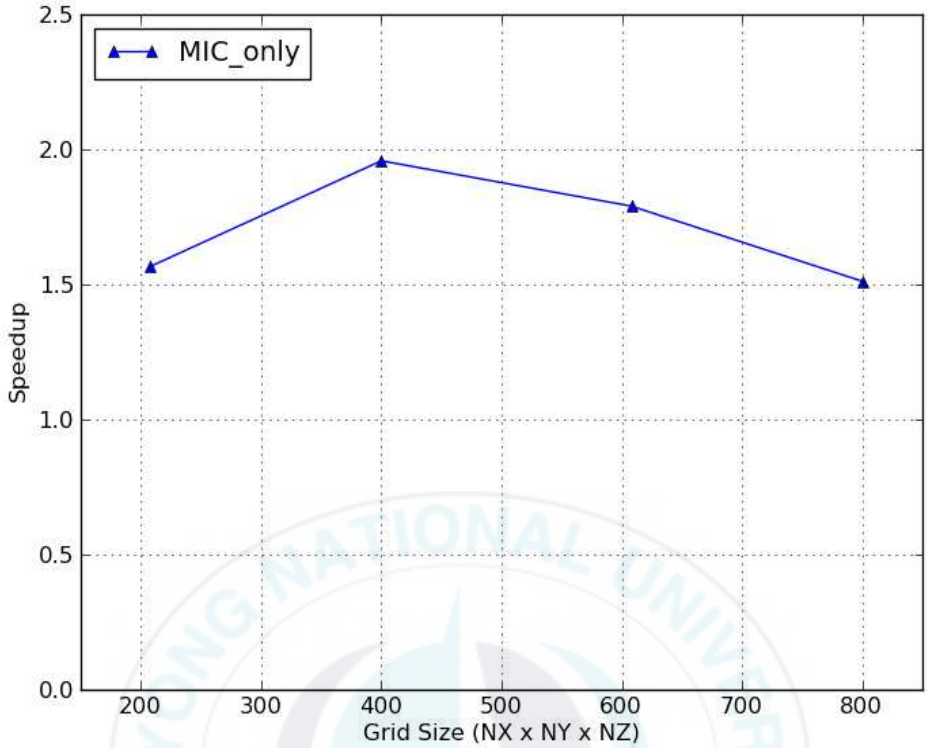


Fig. 4.7. Relative speedups of the Xeon Phi program with respect to the CPU programs using the same model

2차원과 마찬가지로 SEG/EAGE 3차원 암염돔 모델(Aminzadeh et al.,1994)의 적용을 통해 CPU와 Xeon Phi의 연산시간을 비교해 보았다.(Fig. 4.8)

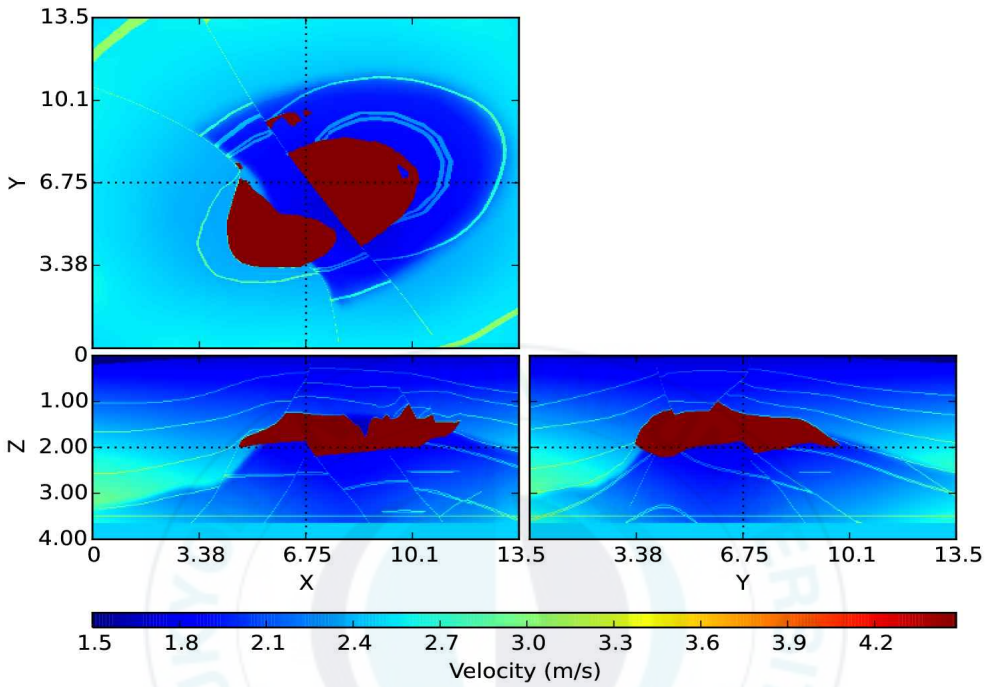


Fig. 4.8. SEG/EAGE salt-dome model(Aminzadeh et al.,1994)

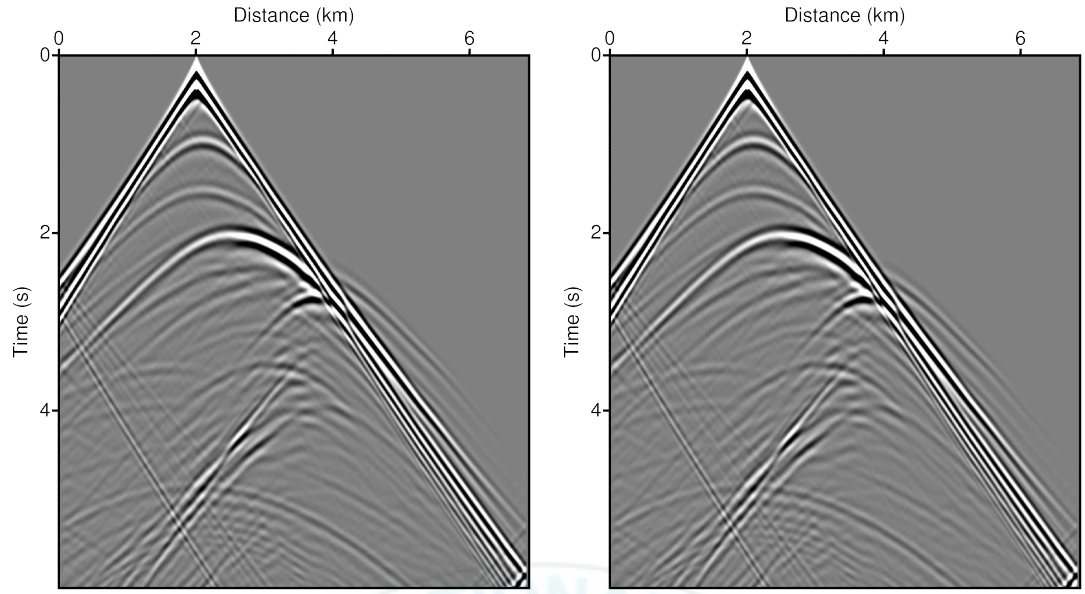
실험에 이용한 모델링 환경은 Table 4.6 와 같고 스레드 분배 방식은 Scatter를 이용하였다. 샘플링 간격은 2ms로 총 6초간 기록하였고 시간에 대해 2차 공간에 대해 8차 유한 차분법을 사용하였다. 경계조건은 병렬화와 최적화된 알고리즘에 맞게 수정한 흡수 경계 조건(Table. 3.4)을 적용하였다.

Table. 4.6 Apply 3D SEG/EAGE salt dome modeling parameter

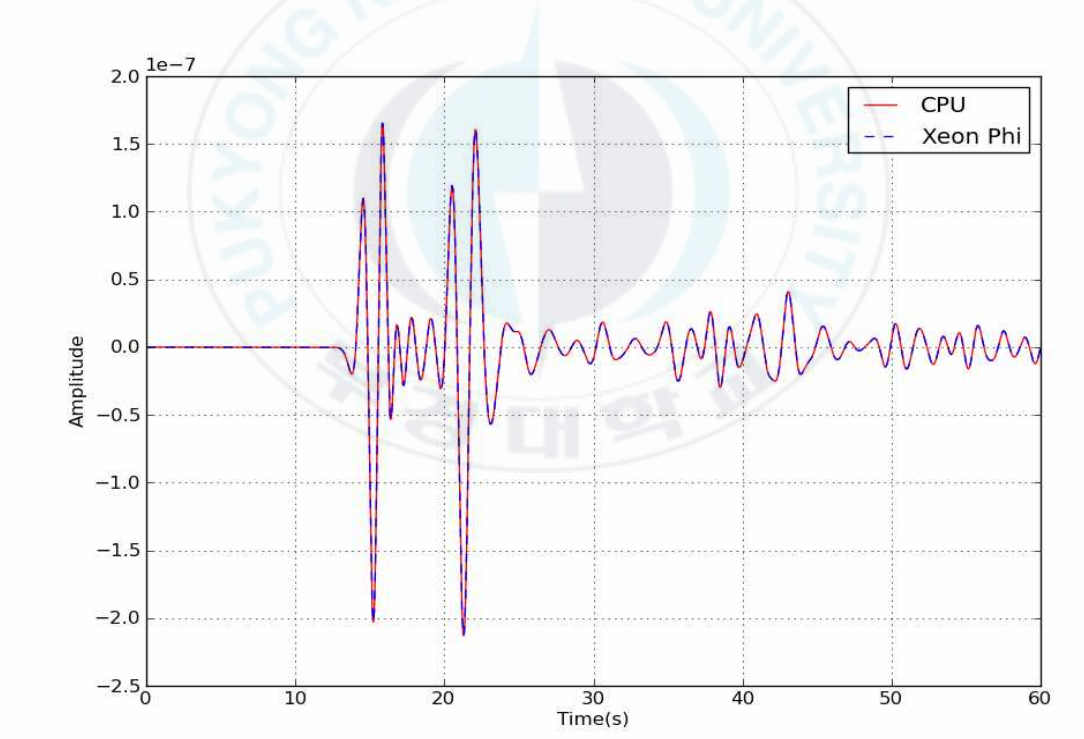
parameter	value
Number of X-Y-Z axis	688,684,209
Number of X-Y-Z Thread block	124,1,2
Grid interval(m)	20
Sampling interval(ms)	2
frequency(Hz)	10

실험 결과, CPU의 계산시간은 146.40초, Xeon Phi 의 계산시간은 80.26초로 3차원 모델링 또한 Xeon Phi 를 사용하여 CPU 대비 54.9% 수준으로 계산시간이 단축됨을 확인 하였다.

Fig 4.9 은 2차원 속도 모델 적용한 것과 같이 CPU와 Xeon Phi에서 각각 모델링을 통해 얻은 파동장과 트레이스를 나타낸 것이다. 트레이스는 송신원에서 2km 떨어진 지점에서 추출하였고 결과를 비교했을 때 파형 및 진폭이 일치함을 나타내 정확도 또한 검증이 되었다.



(a) (b)



(c)

Fig. 4.9 Surface seismogram (y=6km) from (a) CPU and (b) Xeon Phi programs. (c) 'CPU' and 'Xeon Phi' trace curves show the traces from the CPU and Xeon Phi programs.

5. 토의

지금 까지 Xeon Phi와 병렬 프로그래밍을 이용하여 파동방정식 모델링을 구현하고 실험을 통해 CPU 와 Xeon Phi 의 성능을 비교해 보았다. 앞의 성능 비교는 단일 CPU(8코어) 와 Xeon Phi 단일 카드(61코어)을 비교 한 것으로 다중 혹시 멀티 CPU 나 하이퍼 스레딩(Hyper-Threading) 기술을 이용하면 상대적으로 성능(클럭수 등)이 낮은 Xeon Phi 의 속도향상이 이루어지지 않는다. 하지만 Xeon Phi 는 Coprocessor로써 메인 프로세서(CPU) 이외에 수치 연산을 도와주는 가속기 역할을 한다. 본 연구에서는 CPU 와 Xeon Phi 의 정확한 비교를 위해 실행 옵션을 Native로 수행하였지만 Offload 사용하거나 혹은 매니코어의 장점을 살려 반복적인 계산을 CPU와 Xeon Phi 에서 동시에 수행하게 된다면 그 성능은 배가 될 것이다.

또한 Xeon Phi 는 프로그래밍 면에서 같은 알고리즘 코드를 CPU 와 Xeon Phi 에 적용할 수 있는 것이 가장 큰 특징이다. CPU 프로세서와 Xeon Phi 코프로세서용 프로그램을 각각 따로 개발하고 관리할 필요 없이 하나의 프로그램 소스 코드만 개발하고 관리할 수 있다. 이것을 바탕으로 알고리즘 코드를 Xeon CPU 프로세서 혹은 Xeon Phi 둘 중에 한번만 최적화 하면 자연스럽게 둘 다 성능 향상을 얻을 수 있다. 굳이 Xeon Phi 를 기반으로 한 최적화 작업을 수행하고 또 Xeon CPU 를 기반으로 최적화 할 필요 없다. 따라서 개발자가 새로운 프로그램 언어나 환경, 라이브러리 도구들을 배울 필요 없고 이미 익숙하게 사용하고 있는 환경에서 작업을 수행 할 수 있다. 본 연구에서 Xeon Phi 의 이점을 이용하여 기본적인 자료 처리 기술인 파동 전파 모델링을 효율적으로 만들었던 것처럼 역시간 구조보정이나 완전 파형 역산 같은 고해상도 자료 처리에 사용할 수 있다.

4. 결론

본 연구에서는 격자 기반의 유한차분법을 이용한 시간 영역에서의 파동 전파 모델링 시, Xeon Phi 와 병렬 프로그래밍을 통해 효율적인 모델링 방법을 제시하였다. MPI 와 OpenMP 를 통한 2차원 및 3차원 파동 전파 모델링을 구현하고, 송신원 개수와 격자 크기에 따른 CPU 와의 성능을 비교해 봄으로써 Xeon Phi 의 우수함을 알 수 있었다. 매니코어 및 같은 알고리즘을 그대로 적용할 수 있다는 장점을 통해 동시에 CPU 와 Xeon Phi 를 사용하거나 최적화를 통해 더 나은 성능을 얻을 수 있을 것이다. 본 연구에서 Xeon Phi 의 효율적인 사용은 탄성파 자료처리에 있어 계산시간을 줄이고 그에 따른 비용을 절감할 수 있을 것이다. 또한 이를 발전시켜 구조보정, 파형역산 등 고해상도 지구물리자료 처리 및 해석 과정을 향상 시킬 수 있을 것으로 판단된다.

참고문헌

Alford, R., Kelly, K. and Boore, D. M. (1974). "Accuracy of Finite-Difference Modeling of the Acoustic Wave Equation," Geophysics, Vol 39, No6, pp 834-842.

Alterman, Z. and Karal, F. (1968). "Propagation of Elastic Waves in Layered Media by Finite Difference Methods," Bulletin of the Seismological Society of America, Vol 58, No1, pp 367-398.

Aminzadeh, F., Burkhard, N., Nicoletis, L., Rocca, F. and Wyatt, K. (1994). "SEG/EAEG 3-D Modeling Project: 2nd Update," The Leading Edge, Vol 13, No9, pp 949-952.

Banaś, K. and Krużel, F. (2014). "OpenCL performance portability for xeon phi coprocessor and NVIDIA GPUs: A case study of finite element numerical integration," European Conference on Parallel Processing, pp 158-169.

Cédric, A. (2014). "Eight Optimizations for 3-Dimensional Finite Difference(3DFD) Code with an Isotropic(ISO) ," Intel Developer Zone,http://software.intel.com/en-us/articles/eight-eptimizations-for-3-dimensional-finite-difference-3dfd-code-with-an-isotropic-iso

Chapman, B., Jost, G. and Van Der Pas, R. (2008). Using OpenMP: Portable Shared Memory Parallel Programming, MIT press.

Chrysos, G. (2014). "Intel® Xeon Phi™ Coprocessor-the Architecture," Intel Whitepaper, .

Chrysos, G. and Engineer, S. P. (2012). "Intel xeon phi coprocessor

(codename knights corner)," Proceedings of the 24th Hot Chips Symposium, .

Jeffers, J. and Reinders, J. (2013). Intel Xeon Phi Coprocessor High-Performance Programming, Newnes.

Jeffers, J. and Reinders, J. (2014). High Performance Parallelism Pearls: Multicore and Many-Core Programming Approaches.

Kelly, K., Ward, R., Treitel, S. and Alford, R. (1976). "Synthetic Seismograms: A Finite-Difference Approach," Geophysics, Vol 41, No1, pp 2-27.

Keys, R. G. (1985). "Absorbing Boundary Conditions for Acoustic Media," Geophysics, Vol 50, No6, pp 892-902.

Kim, Y., Cho, Y., Jang, U. and Shin, C. (2013). "Acceleration of Stable TTI P-Wave Reverse-Time Migration with GPUs," Comput. Geosci., Vol 52, pp 204-217.

Kreyszig, E. (2010). Advanced Engineering Mathematics, John Wiley & Sons.

Mori, F., Matsumoto, M. and Furumura, T. (2014). "Performance optimization of the 3D FDM simulation of seismic wave propagation on the intel xeon phi coprocessor using the ppOpen-APPL/FDM library," International Conference on High Performance Computing for Computational Science, pp 66-76.

Pacheco, P. (2011). An Introduction to Parallel Programming, Elsevier.

Reynolds, A. C. (1978). "Boundary Conditions for the Numerical Solution of Wave Propagation Problems," Geophysics, Vol 43, No6, pp 1099-1110.

Suh, S. and Wang, B. (2011). "Expanding domain methods in GPU based TTI reverse time migration," 2011 SEG Annual Meeting, .

Tarantola, A. (1984). "Inversion of Seismic Reflection Data in the Acoustic Approximation," Geophysics, Vol 49, No8, pp 1259-1266.

Versteeg, R. (1994). "The Marmousi Experience: Velocity Model Determination on a Synthetic Complex Data Set," The Leading Edge, Vol 13, No9, pp 927-936.

Abstract

In this research, We suggested efficient time-domain wave propagation modeling using parallel programming and Xeon Phi as parallel calculation accelerator. Using parallel programming and many-core characteristics of Xeon Phi can perform modeling more efficiently traditional cpu environments. To confirm this research, we tested the performance of the program by varying the number of source position and the size of grid. And we also compared calculation time and accuracy by applying velocity models. Based on the modeling, it is possible to expect improvements in performance by applying high performance data processing such as waveform inversion, reverse time migration(RTM).

Keywords : Xeon Phi, Accelerator, Wave propagation modeling, Parallel calculation, finite difference method

Student Number : 201555173

부록 프로그램 코드

1. 최적화된 유한 차분법 프로그램(Intel Developer zone - Eight Optimizations for 3-Dimensional Finite Difference(3DFD) Code with an Isotropic(ISO))

```
void iso_3dfd_it(float *ptr_next_base, float *ptr_prev_base, float *ptr_vel_base, float *coeff,
  const int n1, const int n2, const int n3, const int num_threads,
  const int n1_Tblock, const int n2_Tblock, const int n3_Tblock){
  int dimn1n2 = n1*n2;
  int n3End = n3 - HALF_LENGTH;
  int n2End = n2 - HALF_LENGTH;
  int n1End = n1 - HALF_LENGTH;
  #pragma omp parallel OMP_N_THREADS default(shared)
  {
    float* ptr_next;
    float* ptr_prev;
    float* ptr_vel;
    float value;
    int izEnd;
    int iyEnd;
    int ixEnd;
    const int vertical_1 = n1, vertical_2 = n1*2, vertical_3 = n1*3, vertical_4 = n1*4;
    const int front_1 = dimn1n2, front_2 = dimn1n2*2, front_3 = dimn1n2*3, front_4 = dimn1n2*4;
    const float c0=coeff[0], c1=coeff[1], c2=coeff[2], c3=coeff[3], c4=coeff[4];
```

```
    //At this point, we must handle the stencil possible sizes.
#if ( HALF_LENGTH == 8 )
    const int vertical_5 = n1*5, vertical_6 = n1*6, vertical_7 = n1*7, vertical_8
= n1*8;
    const int front_5 = dimn1n2*5, front_6 = dimn1n2*6, front_7 = dimn1n2*7,
front_8 = dimn1n2*8;
    const float c5=coeff[5], c6=coeff[6], c7=coeff[7], c8=coeff[8];
#endif
    __assume_aligned((void*)vertical_1, CACHELINE_BYTES);
    __assume_aligned((void*)vertical_2, CACHELINE_BYTES);
    __assume_aligned((void*)vertical_3, CACHELINE_BYTES);
    __assume_aligned((void*)vertical_4, CACHELINE_BYTES);
    __assume_aligned((void*)front_1, CACHELINE_BYTES);
    __assume_aligned((void*)front_2, CACHELINE_BYTES);
    __assume_aligned((void*)front_3, CACHELINE_BYTES);
    __assume_aligned((void*)front_4, CACHELINE_BYTES);
    //Handle all size of stencil
#if( HALF_LENGTH == 8 )
    __assume_aligned((void*)vertical_5, CACHELINE_BYTES);
    __assume_aligned((void*)vertical_6, CACHELINE_BYTES);
    __assume_aligned((void*)vertical_7, CACHELINE_BYTES);
    __assume_aligned((void*)vertical_8, CACHELINE_BYTES);
    __assume_aligned((void*)front_5, CACHELINE_BYTES);
    __assume_aligned((void*)front_6, CACHELINE_BYTES);
    __assume_aligned((void*)front_7, CACHELINE_BYTES);
    __assume_aligned((void*)front_8, CACHELINE_BYTES);
#endif
    __declspec(align(CACHELINE_BYTES)) float div[n1_Tblock];
```

```
#pragma omp for OMP_SCHEDULE collapse(3)
for(int bz=HALF_LENGTH; bz<n3End; bz+=n3_Tblock){
for(int by=HALF_LENGTH; by<n2End; by+=n2_Tblock){
for(int bx=HALF_LENGTH; bx<n1End; bx+=n1_Tblock){
  izEnd = MIN(bz+n3_Tblock, n3End);
  iyEnd = MIN(by+n2_Tblock, n2End);
  ixEnd = MIN(n1_Tblock, n1End-bx);

  for(int iz=bz; iz<izEnd; iz++) {
  for(int iy=by; iy<iyEnd; iy++) {
    ptr_next = &ptr_next_base[iz*dimn1n2 + iy*n1 + bx];
    ptr_prev = &ptr_prev_base[iz*dimn1n2 + iy*n1 + bx];
    ptr_vel = &ptr_vel_base[iz*dimn1n2 + iy*n1 + bx];
    __assume_aligned(ptr_next, CACHELINE_BYTES);
    __assume_aligned(ptr_prev, CACHELINE_BYTES);
    __assume_aligned(ptr_vel, CACHELINE_BYTES);
    #pragma ivdep
    for(int ix=0; ix<ixEnd; ix++) {
      value = ptr_prev[ix]*c0
        + c1 * (FINITE_ADD(ix, 1)
            + FINITE_ADD(ix, vertical_1)
            + FINITE_ADD(ix, front_1))
        + c2 * (FINITE_ADD(ix, 2)
            + FINITE_ADD(ix, vertical_2)
            + FINITE_ADD(ix, front_2))
        + c3 * (FINITE_ADD(ix, 3)
            + FINITE_ADD(ix, vertical_3)
            + FINITE_ADD(ix, front_3))
```

```
            + c4 * (FINITE_ADD(ix, 4)
                + FINITE_ADD(ix, vertical_4)
                + FINITE_ADD(ix, front_4))
#if( HALF_LENGTH == 8)
            + c5 * (FINITE_ADD(ix, 5)
                + FINITE_ADD(ix, vertical_5)
                + FINITE_ADD(ix, front_5))
            + c6 * (FINITE_ADD(ix, 6)
                + FINITE_ADD(ix, vertical_6)
                + FINITE_ADD(ix, front_6))
            + c7 * (FINITE_ADD(ix, 7)
                + FINITE_ADD(ix, vertical_7)
                + FINITE_ADD(ix, front_7))
            + c8 * (FINITE_ADD(ix, 8)
                + FINITE_ADD(ix, vertical_8)
                + FINITE_ADD(ix, front_8))
#endif
            ;
          ptr_next[ix] = 2.0f* ptr_prev[ix] - ptr_next[ix] + value*ptr_vel[ix];
        }// end x
      }}//end Y and Z
    }}}//end of cache blocking
  }//end of parallel
}//end of function
```

2. 파동 방정식 모델링 프로그램

```
void iso_3dfd_wave(float *ptr_next, float *ptr_prev, float *ptr_vel, float *ptr_vdt2, float *coeff, const TYPE_INTEGER n1, const TYPE_INTEGER n2, const TYPE_INTEGER n3, const TYPE_INTEGER num_threads, const TYPE_INTEGER nreps, const TYPE_INTEGER n1_Tblock, const TYPE_INTEGER n2_Tblock, const TYPE_INTEGER n3_Tblock)
{
        int sz=HALF_LENGTH+2,sy=201,sx=201;
        int isrc=n1*n2*sz+n1*sy+sx;
        int rz=sz;
        int ry=sy;
        int offset=n1*n2*rz+n1*ry;
        char buf[200];
        printf("pwd=%s\n",getcwd(buf,200));
        float *w;
        w=(float*)malloc(sizeof(float)*nreps);
        rickerf(w,10.,DT,nreps);
  FILE *fp=fopen("seismo.bin","wb");
  float *tmp;
  for(TYPE_INTEGER it=0; it<nreps; it++){
          if( it%100 == 0)
                  printf("it=%d\n",it);
    iso_3dfd_it(ptr_next, ptr_prev, ptr_vdt2, coeff,
                n1, n2, n3, num_threads, n1_Tblock, n2_Tblock, n3_Tblock);
    ptr_next[isrc]+=ptr_vdt2[isrc]*w[it];
// here's where boundary conditions happen
    iso_3dfd_bc_omp(ptr_next, ptr_prev, ptr_vel, n1, n2, n3, num_threads, n1_Tblock, n2_Tblock, n3_Tblock);

    fwrite(&ptr_next[offset],sizeof(float),n1,fp);

    tmp=ptr_next;
    ptr_next=ptr_prev;
```

```
    ptr_prev=tmp;
  } // time loop
  fclose(fp);
  free(w);
}
void iso_3dfd_bc_omp(float *ptr_next,  float *ptr_prev,  float *ptr_vel, const TYPE_INTEGER n1, const TYPE_INTEGER n2, const TYPE_INTEGER n3, const TYPE_INTEGER num_threads, const TYPE_INTEGER n1_Tblock, const TYPE_INTEGER n2_Tblock, const TYPE_INTEGER n3_Tblock)
{
#pragma omp parallel OMP_N_THREADS{
        const TYPE_INTEGER dimn1n2 = n1*n2;
        TYPE_INTEGER n3End = n3 - HALF_LENGTH;
        TYPE_INTEGER n2End = n2 - HALF_LENGTH;
        TYPE_INTEGER n1End = n1 - HALF_LENGTH;
        const float dtoh=DT/DXYZ;
// right
        #pragma omp for collapse(2)
        for(TYPE_INTEGER bz=HALF_LENGTH; bz<n3End; bz+=n3_Tblock)
        for(TYPE_INTEGER        by=HALF_LENGTH;         by<n2End; by+=n2_Tblock){
                TYPE_INTEGER izEnd = MIN(bz+n3_Tblock, n3End);
                TYPE_INTEGER iyEnd = MIN(by+n2_Tblock, n2End);
                TYPE_INTEGER offset,i;
                for(TYPE_INTEGER iz=bz; iz<izEnd; iz++)
                for(TYPE_INTEGER iy=by; iy<iyEnd; iy++){
                        offset = iz*dimn1n2 + iy*n1 ;
                for(TYPE_INTEGER ix=n1End; ix<n1; ix++){
                                i=offset+ix;
ptr_next[i]=ptr_vel[i]*dtoh*(ptr_prev[i-1]-ptr_prev[i])+ptr_prev[i];
                        }
                }
        }
// left
```

```
        #pragma omp for collapse(2)
        for(TYPE_INTEGER bz=HALF_LENGTH; bz<n3End; bz+=n3_Tblock)
        for(TYPE_INTEGER by=HALF_LENGTH; by<n2End;
by+=n2_Tblock){
                TYPE_INTEGER izEnd = MIN(bz+n3_Tblock, n3End);
                TYPE_INTEGER iyEnd = MIN(by+n2_Tblock, n2End);
                TYPE_INTEGER offset,i;
                for(TYPE_INTEGER iz=bz; iz<izEnd; iz++)
                for(TYPE_INTEGER iy=by; iy<iyEnd; iy++){
                        offset = iz*dimn1n2 + iy*n1 ;
                for(TYPE_INTEGER ix=HALF_LENGTH-1; ix>=0; ix--){
                        i=offset+ix;
        ptr_next[i]                                                    =
ptr_vel[i]*dtoh*(ptr_prev[i+1]-ptr_prev[i])+ptr_prev[i];
                        }
                }
        }
// back
        #pragma omp for collapse(2)
        for(TYPE_INTEGER bz=HALF_LENGTH; bz<n3End; bz+=n3_Tblock)
        for(TYPE_INTEGER bx=0; bx<n1; bx+=n1_Tblock){
                TYPE_INTEGER izEnd = MIN(bz+n3_Tblock, n3End);
                TYPE_INTEGER ixEnd = MIN(n1_Tblock, n1-bx);
                TYPE_INTEGER offset,i;
                for(TYPE_INTEGER iz=bz; iz<izEnd; iz++)
                for(TYPE_INTEGER iy=n2End; iy<n2; iy++) {
                        offset = iz*dimn1n2 + iy*n1 + bx;
                for(TYPE_INTEGER ix=0;ix<ixEnd; ix++){
                        i=offset+ix;
        ptr_next[i]                                                    =
ptr_vel[i]*dtoh*(ptr_prev[i-n1]-ptr_prev[i])+ptr_prev[i];
                        }
                }
        }
```

```
// front
        #pragma omp for collapse(2)
        for(TYPE_INTEGER bz=HALF_LENGTH; bz<n3End; bz+=n3_Tblock)
        for(TYPE_INTEGER bx=0; bx<n1; bx+=n1_Tblock){
                TYPE_INTEGER izEnd = MIN(bz+n3_Tblock, n3End);
                TYPE_INTEGER ixEnd = MIN(n1_Tblock, n1-bx);
                TYPE_INTEGER offset,i;
                for(TYPE_INTEGER iz=bz; iz<izEnd; iz++)
                for(TYPE_INTEGER iy=HALF_LENGTH-1; iy>=0; iy--) {
                        offset = iz*dimn1n2 + iy*n1 + bx;
                for(TYPE_INTEGER ix=0;ix<ixEnd; ix++){
                        i=offset+ix;
        ptr_next[i]=ptr_vel[i]*dtoh*(ptr_prev[i+n1]-ptr_prev[i])+ptr_prev[i];
                        }
                }
        }
// bottom
        for(TYPE_INTEGER iz=n3End; iz<n3; iz++){
        #pragma omp for collapse(2)
        for(TYPE_INTEGER by=0; by<n2; by+=n2_Tblock)
        for(TYPE_INTEGER bx=0; bx<n1; bx+=n1_Tblock){
                TYPE_INTEGER iyEnd = MIN(by+n2_Tblock, n2);
                TYPE_INTEGER ixEnd = MIN(n1_Tblock, n1-bx);
                TYPE_INTEGER offset,i;
                for(TYPE_INTEGER iy=by; iy<iyEnd; iy++) {
                        offset = iz*dimn1n2 + iy*n1 + bx;
                for(TYPE_INTEGER ix=0;ix<ixEnd; ix++){
                        i=offset+ix;
ptr_next[i]= ptr_vel[i]*dtoh*(ptr_prev[i-dimn1n2]-ptr_prev[i])+ptr_prev[i];
                        }
                }
        }
}
}//omp
```