기술 보고서

2014년 나노 및 기상분야 계산가속기 이식성 및 성능테스트 보고서

2014.11

슈퍼컴퓨팅사용자지원실

목 차

개요 및 목록 ····· 1
1. VASP ····· 2
2. Quantum Espresso ····· 27
3. WRF ····· 37

2014년도 나노 및 기상분야 계산가속기 이식성 및 성능테스트 보고서

○ 개요

- 슈퍼컴퓨터 4호기에서 빈번하게 사용되는 응용코드인 VASP(나노물리), Quantum Expresso(나노물리), WRF(기상)을 계산가속기 기반의 초고성능시스템에서 설치방법, 이식성 및 성능테스트를 수행

순위	프로그램명	분야	비고	가속기 지원 테스트
1	VASP	나노물리/양자화학	상용	Xeon Phi: Native/AO NVIDIA GPU: CUDA 구현물 (비공개)
8	Quantum Espresso(QE)	나노물리/양자화학	오픈소스	Xeon Phi: Native/AO NVIDIA GPU: CUDA 구현물
10	WRF	대기과학	오픈소스	Xeon Phi: Native/Symmetric NVIDIA GPU: OpenACC 구현물

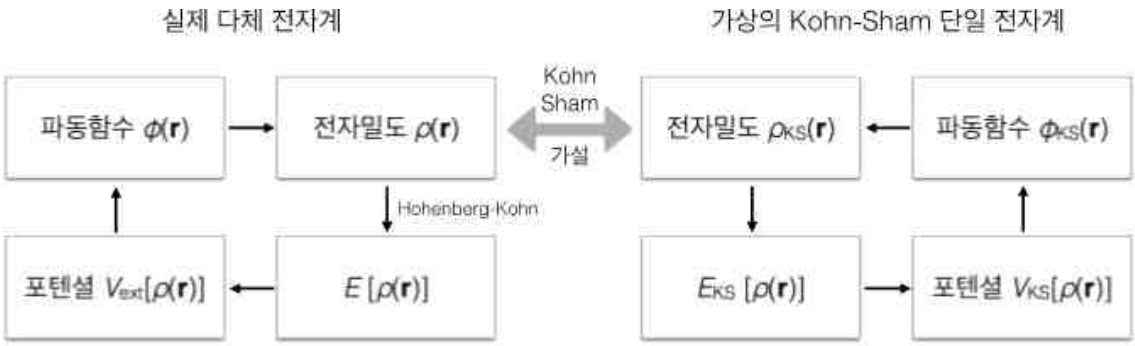
- 55.734Tflops 급의 가속기 기반 계산환경 테스트베드에서 테스트: Intel Xeon Phi 7120p 18개(1.238Tflops/1ea*18ea=22.28Tflops)와 Tesla K40m 15개(1.43 Tflops/1ea*15ea=21.45Tflops)

기반 계산환경 테스트베드 사양

구분		사양		
제조사/시스템명		IBM Nextscale nx360M4		
노드수		30		
구분		프로세서	가속기	
모델명		Intel Xeon E5-2670 v2	Intel Xeon Phi 7120P	NVIDIA Tesla K40m
클럭(GHz)		2.5	1.238	1.43
CPU Core수 (개)	소켓 당	10	61	2,880
	노드 당	20	61	2,880
	전체	600		
이론성능 (GFlops)	노드 당	400	1,208(15노드)	1,430(15노드)
	시스템 전체	12,000	22,284(3노드는 카드2개, 12노드는 카드1개)	21,450
		55,734		
노드 당 소켓/가속기 수 (개)		2	1	1
메모리		DDR3 1,866MHz 128GB (16GBx8ea)	GDDR5 5.5GHz 16GB	GDDR5 12GB
내장디스크		SSD 256GB x 1ea		
Interconnection 네트워크		IB 4x FDR(56Gbps)		

1. VASP

- 개요
  - Vienna ab-initio Simulation Package(VASP)은 밀도 범함수(Density Functional Theory) 이론에 기반한 제일원리 전자구조 상용 코드로서 분자 및 결정의 전자 구조 및 물성을 계산함
  - 슈퍼컴퓨터 4호기 타키온 2차 시스템에서 in-house코드를 제외하고 가장 빈번하게 사용하는 응용으로 가속기 시스템이 향후 도입될 경우에 이식성과 성능이 중요함
  - Intel Xeon Phi는 공식적으로 지원하지 않으며 NVIDIA GPU은 Vienna 대학을 중심으로 현재 CUDA 코드(VASP 5.3.4)로 개발중임
- VASP 소개
  - 밀도범함수 이론은 많은 수의 전자와 원자 핵이 쿨롱 힘을 통해 서로 상호작용하는 복잡한 계에서의 바닥 상태(ground state)를 계산하기 위한 방법으로서, 많은 수의 전자를 동시에 고려해야 하는 다체계(many-particle system) 위에서의 문제를 단일입자계 (single-particle system) 위의 문제로 치환
  - 이를 통해서, 수십~수백 개의 원자들을 포함하는 계들을 양자역학을 통해 다룰 수 있게 한다. 밀도범함수 이론을 통해서, 주어진 시스템 또는 물질을 모델링하기 위한 별도의 변수 없이 물질의 결정 구조와 원자 조성만으로 물성을 계산할 수 있음. 따라서, 밀도범함수 이론을 사용한 코드들은 제일원리 전자구조계산(first-principles electronic structure calculation) 코드로 불리기도 함
  - 밀도범함수 이론 보다 구체적으로, 밀도범함수 이론은 전자의 바닥 상태 밀도 분포함수로부터 해당 물질의 모든 바닥 상태의 성질을 계산할 수 있음을 말하는 Hohenberg-Kohn 정리, 그리고 정확한 바닥 상태의 전자 밀도를 원래의 다체계 문제와 똑같이 재현할 수 있는 단일 입자계의 존재를 가정하는 Kohn-Sham 가설에 기반하고 있음
  - 또한 Hohenberg-Kohn 정리는 주어진 전자 밀도로부터 해당 계의 운동 에너지 및 상호 작용 에너지를 정확하게 계산할 수 있는 밀도 범함수(functional)의 존재를 말하고 있기 때문에, 일단 Kohn-Sham 가설을 통해 쉽게 풀 수 있는 단일 입자 문제로 바뀐 문제를 풀어 정확한 계의 바닥 상태 전자 밀도를 얻을 수 있다면, 그로부터 원래 다체계의 에너지를 정확하게 계산할 수 있을 것이라 기대할 수 있음
  - 단 슈뢰딩거 방정식 자체가 전자의 밀도, 즉 방정식을 풀어서 나온 전자의 거동에 의존하기 때문에, 이 단일 입자 슈뢰딩거 방정식(또는, Kohn-Sham 방정식)을 풀기 위해서는 방정식을 푼 결과를 방정식에 되먹임해 가며 도출된 결과가 입력된 결과 사이의 차이를 좁혀 가는, 즉 자기 모순이 없는(self-consistent) 풀이법을 사용해야 함 (아래 그림 참조)



Kohn-Sham 이용한 밀도범함수 코드 계산의 모식도

- 슈뢰딩거 방정식의 행렬 풀이 슈뢰딩거 방정식은 기본적으로 3차원 공간 위에서의 편미분 방정식이지만, 적절한 기저(basis)를 이용하여 문제를 행렬의 eigenvalue와 eigenvector를 구하는 것으로 치환할 수 있음. 이 때, 문제를 정의하는 행렬 부분, 또는 원래 슈뢰딩거 방정식에서의 연산자 부분은 '해밀토니안'(Hamiltonian)으로 불리며, 우리가 풀고자 하는 계에 대한 정보를 담고 있음
- 해밀토니안 행렬을 대각화해서 얻은 전자의 eigenvector, 또는 wavevector는 각 전자의 '상태 함수'(state function)로 해석되며, 각각의 상태 함수에 대응되는 eigenvalue 값은 각 상태가 갖는 '특성 에너지'(eigenenergy)로 해석
- 주어진 계의 상태 함수과 특성 에너지들로부터 해당 계의 바닥 상태를 만들어 낼 수 있으며, 또한 바닥 상태 함수를 통해 해당 계의 전자 밀도를 만들어 낼 수 있음. 이 전자 밀도를 이용해 해밀토니안 행렬을 다시 만들어 내어 문제를 자기 모순 없는 방식으로 풀어 나가게 됨
- 전자 밀도범함수에 대해 전자의 밀도로부터 해밀토니안을 만들어내는 것은, Kohn-Sham 가설에 의한 방법론 안에서는, 개개의 전자들이 느끼는 쿨롱 포텐셜 함수를 전자의 밀도에 의존하는 범함수 형태로 쓰는 방법을 사용
- 단순한 전자-전자 밀도 분포 사이의 고전적인 쿨롱 포텐셜 외에도, 파울리의 배타 원리에서 유래하는 양자역학적 exchange-correlation 에 의한 효과 역시 이 밀도 범함수에 의해 구현되어야 하기 때문에, 이 밀도 범함수의 형태를 보다 정확하게 정하는 것은 밀도범함수 이론의 핵심적인 부분
- 하지만 이는 매우 어려운 작업이며, 아직까지는 전자 계의 전자-전자 상호작용을 모든 물질에서 잘 기술하는 범함수의 형태는 밝혀지지 않은 상태임
- 현재까지는 국소 밀도 근사법(local density approximation, LDA) 또는 generalized gradient approximation(GGA)을 통해 구한 범함수의 형태가 널리 사용됨. 또한 전자-전자 상호작용이 강한 전이 금속 물질 또는 정량적으로 정확한 값이 필요한 물질들의 계산에 대해서는 밀도범함수 이론 코드 위에 상호작용을 다루기 위한 다른 방법론들을 추가한 것들을 사용. 이들

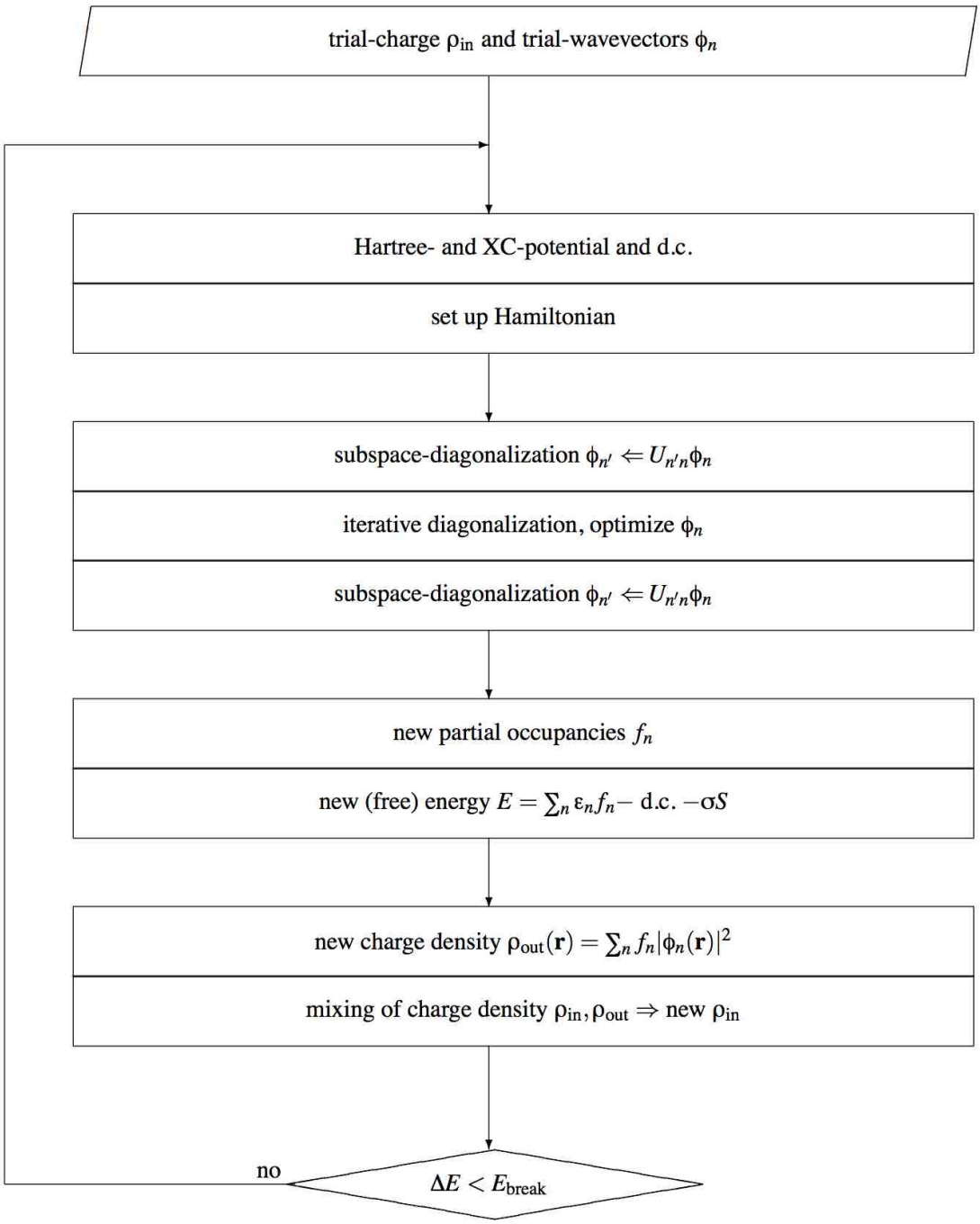
중 일부 방법론(DFT+U, GW, hybrid functional)들은 VASP에 구현되어 있음

- 평면파 기저 VASP은 기저로서 평면파 기저(plane-wave basis), 또는 원자들의 코어 상태를 다루기 위해 프로젝터에 의해 보강된 평면파 기저(projector-augmented wave, PAW)를 사용. 이는 결정 구조와 같이 일정한 공간에서의 주기성을 갖는 물질들에 대해서 효율적인 선택.
- 또한, Gaussian과 같이 공간적으로 국소화된 다른 기저를 사용하는 코드들에 비해 의존 변수의 수가 줄어든다는 장점. 다만, 공간적으로 국소화된 원자들에 위치한 전자들을 설명하기 위해서는 많은 수의 평면파들이 필요하게 되어 행렬 대각화 문제를 푸는데 필요한 시간이 크게 증가
- 여기에서 계산 시간을 줄이기 위해 적용할 수 있는 방법은 두 자기로 나누어 생각할 수 있는데, 하나는 계산에 필요한 평면파의 수를 줄이는 방법과, 다른 하나로는 행렬 대각화 문제를 최적화하는 방법이 있음
- 첫번째 문제에는 개개의 원자들의 국소화된 코어 상태를 근사하는 '유사포텐셜'(pseudopotential) 방법을 적용하며, 두번째 문제에는 주어진 행렬에서 필요한 낮은 에너지 상태들의 eigenvalue/vector들만을 구하는 반복적 대각화 방법(iterative diagonalization)을 사용
- 유사포텐셜 방법 개개의 원자들에 속한 전자들은, 화학적인 관점에서 보면 크게 두 부분. 하나는, 에너지가 낮은 상태를 차지하기 때문에 외부의 변화에 둔감하여 화학적 반응에 참여하지 않는 코어 전자들이고, 다른 하나는 상대적으로 높은 에너지를 가지며 화학 반응에 활발히 참여하는 최외각 전자(valence electron)
- 이 중, 코어 전자들은 매우 국소화된 분포를 갖고 있기 때문에 이들을 평면파로 표현하기 위해서는 매우 많은 수의 평면파 기저가 필요하지만, 이들은 화학결합에 거의 참여하지 않기 때문에 물질의 물성을 다룰 시에는 중요한 영향을 끼치지 않음
- 유사포텐셜 방법은, 이러한 코어 전자 상태들과 또한 원자 핵에서 오는 강한 쿨롱 포텐셜을 묶어 상쇄시키며, 또한 최외각 전자들이 보기에는 원래 원자핵과 코어 전자들과 같은 포텐셜을 주는 포텐셜의 '대체품'을 만들어 내어 계산에 사용하는 방법
- 현재 쓰이는 대부분의 평면파 또는 PAW 기저를 사용하는 밀도범함수 코드들은 유사포텐셜 방법을 사용
- 반복적 대각화 방법 유사포텐셜 방법을 통해 사용되는 기저의 수를 줄인다 하더라도, 개개의 물질들에서 원자 주변의 전자 분포를 기술하는데 필요한 평면파 또는 PAW 기저의 수는 가장 간단한 시스템에서도 수백개를 넘으며, 보다 현실적인 문제들에서는 이 수는 수만에서 수십만개까지 커질 수 있음

- 해밀토니안 행렬을 이러한 큰 공간에서 직접 대각화하는 것은 효율적이지 못하며, 평면파 기저 공간에서 기술된 eigenfunction들과 eigenenergy들은 물리/화학적 관점에서 직관적으로 이해하기도 어렵움. VASP에서는 대신, 평면파 기저들로 전개될 수 있는 일정 수의 시도함수(trial function)들을 정하고, 해밀토니안 행렬을 이 함수들에 의해 기술되는 subspace에서 대각화 한 뒤, 개개의 시도함수들의 에너지 기대값을 낮추는 방향으로 시도함수를 전자들의 바닥상태 함수에 가깝게 최적화
- 시도함수들이 자기모순없는 상태에 도달하면, 다시 말해 결과가 수렴하게 되면, 시도함수들의 eigenenergy는 계의 상태 함수들의 에너지와 일치하며 또한 시도함수들은 계의 바닥상태 함수들과 일치하게 됨
- 시도함수를 사용한다는 점에서는 VASP과 다른 국소적 기저들을 사용하는 코드들과 공통점이 있다고 할 수 있지만, VASP은 계산 과정에서 시도함수가 최적화되기 때문에, 특정한 기저 선택에 결과가 구애받지 않는 보다 신뢰할 수 있는 결과를 얻을 수 있다는 차이점이 있음
- 또한, 해밀토니안 행렬의 대각화는 시도함수들에 의해 기술되는 subspace에서 이루어지기 때문에, 대각화해야 하는 행렬의 크기가 줄어들어 보다 효율적인 계산이 가능하게 됨
- VASP 계산 흐름 VASP 코드의 전체적인 실행 흐름은 다음 페이지 그림에 설명되어 있음. 전체적으로 보면, 코드는 전자 밀도를 구한 후 이를 해밀토니안에 되먹임하여 자기 모순 없는 결과에 접근해 가는 바깥 루프와, 바깥 루프 안쪽에서 해밀토니안이 기술되는subspace의 시도함수들을 최적화하는 안쪽 루프로 구성되어 있음. VASP 코드를 실행할 때 나오는 standard output, 또는 이를 파일에 출력하는 OSZICAR 파일에서 알 수 있음. 출력 형식은 다음과 같음

```
...
entering main loop
       N        E                     dE             d eps       ncg     rms          rms(c)
DAV:   1    0.284788808340E+03    0.28479E+03   -0.16712E+04  2464   0.134E+03
DAV:   2   -0.143379026546E+02   -0.29913E+03   -0.29230E+03  3002   0.373E+02
DAV:   3   -0.381778635987E+02   -0.23840E+02   -0.23282E+02  3196   0.102E+02
DAV:   4   -0.390206722582E+02   -0.84281E+00   -0.84169E+00  4202   0.191E+01
DAV:   5   -0.390397206195E+02   -0.19048E-01   -0.19044E-01  3216   0.227E+00  0.148E+01
DAV:   6   -0.330167909892E+02    0.60229E+01   -0.39322E+01  3734   0.516E+01  0.636E+00
DAV:   7   -0.326661927384E+02    0.35060E+00   -0.41221E+00  3780   0.198E+01  0.267E+00
...
```

- 여기에서 *N*은 바깥 루프의 반복 수, 그리고 *ncg*는 각 바깥 루프마다 시도 함수를 최적화하기 위한 conjugate gradient 단계의 수, 그리고 첫번째 칼럼의 *DAV*는 Davidson iterative diagonalization 방법이 사용되고 있음을 의미. 물론, 결정 구조를 최적화하는 계산을 수행할 때는, 여기에 원자의 구조를 조금씩 최적화해가며 그 때마다 바닥 상태를 새로 계산하는 가장 바깥 쪽의 루프가 추가됨



VASP 개략적인 flow chart (출처 : VASP User Guide)

• 수행 내용

- NVIDIA GPU 벤치마크

* VASP 버전은 5.3.4

* 라이센스 문제로 NVIDIA에서 직접 수행하였으며 다음 물질에 대해서 계산: SrTiO3(6x6x3크기로 수행하였으며 원자수는 530개)

※ 해당 물질은 DFT 계산을 심각하게 하는 그룹에서 사용하는 상당히 규모 있는 원자수

※ 성능향상을 위해서 LREAL옵션을 True로 설정하였음

* CPU 수행시 npar를 노드 수로 설정하고, nsim은 32로 설정하였음

※ nsim은 16,32,48에서 32가 가장 좋은 성능을 보여줌

- Intel Xeon Phi 벤치마크

* Xeon Phi에 대해서 공식적으로 지원하지 않아서 쓰레드 수를 늘여가며 CPU대비 Native 및 AO(automatic Offload) 방식 수행시 성능 검토

* VASP 버전은 5.3.3, 컴파일러는 Intel Composer XE 2014

* 특히 VASP은 MKL의 BLAS(ZGEMM, DGETRF, DPOTRF, DGEQRF)와 FFT 모듈을 이용해서 많은 계산을 수행하므로 AO의 적용가능성 판단에 적합함

* 서로 다른 크기의 단위 격자를 가지는 2가지 물질을 계산하고, 계산에 걸리는 시간을 측정: $SrIrO_3$ bulk(단위 격자의 원자수가 7개인 작은 계), GaTa4Se8 bulk(단위 격자 내의 총 원자 수가 52개)

- SrIrO3 bulk

단위 격자에 Ir 원자 1개, Sr원자 3개, 그리고 산소 원자 3개가 포함되는 작은 계

사용된 변수는

· Monkhorst-Pack k-point grid : 12 x 12 x 12

· Plane-wave energy cutoff : 400 eV

계산에서는 약 1100개의 평면파 기저, 그리고 22개의 시작함수가 사용

총 16번 charge density self-consistency iteration을 거친 후 결과값이 수렴

각 step마다 평균적으로 3400여 번 conjugate gradient step을 거친 후 시작함수가 최적화

- GaTa4Se8 bulk

단위 격자에 Ga 원자4개, Ta원자 16개, 그리고 Se 원자 32개가 포함되어, 단위 격자 내의 총 원자 수는 52개

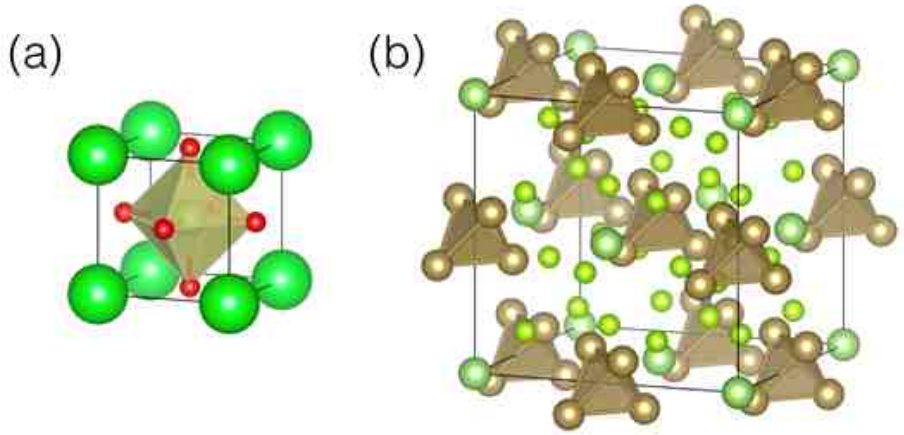
사용된 변수는

· Monkhorst-Pack k-point grid : 4 x 4 x 4

· Plane-wave energy cutoff : 400 eV

계산에서는 약 157000개의 평면파 기저, 그리고 170개의 시작함수가 사용

총 20번의 charge density self-consistency iteration을 거친 후 결과값 수렴
각 step마다 평균적으로 1700여 번의 conjugate gradient step을 거친 후 시작함수가 최적화



(a) SrirO3 격자 구조. 초록색 구가 Sr, 8면체 중앙의 원자가 Ir, 그리고 붉은 작은 구가 O (b) GaTa4Se8의 conventional cubic 단위 격자 구조. 보다 큰 초록색 구가 Ga, 작은 연두색 구가 Se, 그리고 갈색4면체를 이루고 있는 원자들이 Ta

* 계산 시스템 정보 테스트에 사용된 CPU와 Xeon Phi 카드에 대한 정보
· CPU : Intel® XeonTME5-2690v2:10core,3.00GHz,25MCache (계산 노드 당 2개의CPU : 노드 당 core 수는 20개)
· Xeon Phi : Intel® Xeon Phi Coprocessor 7120P: 61cores, 1.24GHz, 30MCache, 16GB메모리

• 설치 방법
- VASP은 LAPACK/BLAS와 FFTW 라이브러리를 지원하는 대부분의 Linux/Unix 기반 시스템에서 컴파일 및 실행될 수 있음
- CPU 설치 방법
* VASP 소스 파일의 압축을 풀면, 다음 2개 디렉토리가 생성됨
./vasp.5.3
./vasp.5.lib
* vasp.5.lib 로 들어가서
cp makefile.linux_ifc_P4 makefile
* 명령을 실행하여 makefile 파일을 준비한 뒤 makefile의 FC 옵션을 ifort로 고친 후 make 명령어를 실행. 정상적으로 컴파일이 종료되면 libdmy.a 파일이 만들어지며, VASP 코드를 컴파일시 라이브러리를 사용함
* 다음으로 vasp.5.3 디렉토리로 이동하여 본 코드를 컴파일. 인텔 컴파일러

를 사용할 시에는, LAPACK, BLAS, 그리고 FFTW 라이브러리 역시 Intel MKL을 사용. 기본 라이브러리들을 컴파일 할 때처럼, 먼저 각 시스템에 맞게 미리 준비된 makefile을 복사한 뒤, 컴파일러 옵션 들 중 아래 항목들을 다음과 같이 수정 후 make 명령어를 통해 코드를 컴파일. 컴파일 전, 사전에 컴파일러 경로에 있는 경로 설정용 스크립트를 실행해 주요 환경변수들을 인식시킴.

* "ulimit - s umlimited" 명령을 통해 VASP이 구동되기 전 메모리 스택의 크기를 늘려 놓아야 문제 없이 실행 가능
* LAPACK 옵션에 대해서 static linking의 경우 다음과 같이 설정

```
FC    = ifort -openmp -mkl
CPP   = $(CPP_) -DMPI  -DHOST=\"LinuxIFC\" -DIFC \
        -DPGF90 -Dkind8 -Davoidalloc \
        -DRPROMU_DGEMV  -DRACCMU_DGEMV \
        -DMPI_BLOCK=8000 \
        -DCACHE_SIZE=16000 -Duse_collective \
        -DNGZhalf
FLAG            = -O3 -ip -xavx -I$(MKLROOT)/include
MKL_FFTW_PATH = $(MKLROOT)/interfaces/fftw3xf/
LAPACK          = -Wl,--start-group\
                  $(MKLROOT)/lib/intel64/libmkl_intel_lp64.a \
                  $(MKLROOT)/lib/intel64/libmkl_core.a \
                  $(MKLROOT)/lib/intel64/libmkl_intel_thread.a \
                  -Wl,--end-group -lpthread -lm
```

* Dynamic linking의 경우는 Intel MKL Link Line Advisor를 참고

- Xeon Phi에서 Native 설치 및 실행 방법

* CPU 환경을 위한 방법과 유사하나, 다음과 같은 차이점이 있음
  · 컴파일 되는 바이너리가 MIC 아키텍처에서 구동될 것이라는 사실을 알려주는 다음과 같이 "-mmic" 옵션이 추가로 필요

```
FC=ifort -openmp -mmic
```

  · 코드가 실행되기 위해 필요한 라이브러리들이 Xeon Phi 카드 내 메모리 공간, 또는 카드가 접근할 수 있는 NFS 경로 상에 존재해야 하며, 일부 라이브러리들(MPI 등)은 Xeon Phi를 위해 재 컴파일되어야 함
  · Intel Composer XE 2013 이후 버전은 Xeon Phi를 위해 컴파일된 MKL 라이브러리들을 제공하며 (FFTW는 사용자가 직접 소스 코드를 컴파일해야 함), 따라서 VASP makefile에는 해당 라이브러리들을 다음과 같이 링크해야 함

```
...
MKL_FFTW_PATH = $(MKLROOT)/interfaces/fftw3xf/
FFT3D          = fftw3d.o  fft3dlib.o \
                 $(MKLROOT)/lib/mic/libfftw3xf_intel.a
LAPACK         = -Wl,--start-group \
                 $(MKLROOT)/lib/mic/libmkl_intel_lp64.a \
                 $(MKLROOT)/lib/mic/libmkl_core.a \
                 $(MKLROOT)/lib/mic/libmkl_intel_thread.a \
                 -Wl,--end-group -lpthread -lm
LIB            = ../vasp.5.lib_mic/libdmy.a \
                 ../vasp.5.lib_mic/linpack_double.o \
                 $(LAPACK)
...
```

· 해당 라이브러리들은 모두 Xeon Phi 상의 메모리에 복사가 되었거나, 또는 Xeon Phi카드 위에서 접근할 수 있는 NFS 경로 위에 존재해야 함

* 실행 방법

· 다음 명령어를 통해 Xeon Phi 카드가 사용할 쓰레드 수를 알려줌

```
export OMP_NUM_THREADS=…
```

· VASP 입력 변수가 있는 디렉토리로 이동 후 다음 명령어를 실행

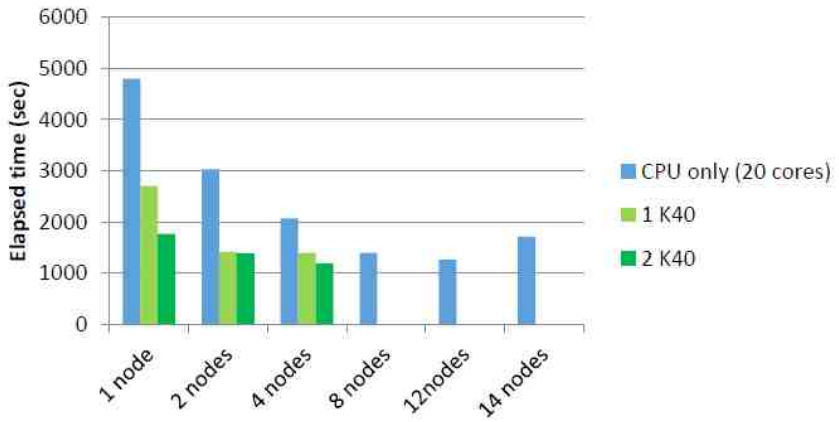
```
./vasp
```

• 결과

- NVIDIA GPU 테스트

* 단일노드에서 CPU(20 cores)대비 K40m 카드 한 장을 사용했을 경우에 약 1.77배, 두 장을 사용했을 경우에 2.71배 성능향상

* 노드 4개 이상에서는 병렬확장성을 보여주지 않음



VASP GPU에서 테스트

- Intel Xeon Phi 테스트

* 단일 노드에서 쓰레드 수를 조정해가면서 Native로 수행가능 했으며 MPI 병렬 작업시 에러 발생

※ Xeon Phi + MPI 환경에서의 오류: CPU에서와 마찬가지로, 단일 카드 내에서의 OpenMP를 이용한 병렬화 성능뿐만 아니라 여러 장의 Xeon Phi 카드를 MPI를 통해 동시에 사용했을때 병렬화 성능을 시험해야 하지만, 아래 문제로 인해서 Xeon Phi 환경에서 MPI를 이용한 병렬화 계산 시험이 이루어지지 못함

· VASP 메인 코드에는, MPI 통신과 관련된 부분 중 각 노드 별 메모리 allocation 및 통신에서의 오류들이 일정 부분 남아 있고, 배열 메모리 접근 오류를 엄격히 확인하는 "-check bound" 컴파일러 옵션을 사용할 시 해당 부분들에서 컴파일이 더 이상 진행되지 않음

· 다만 CPU 환경에서는, 배열 주소에 잘못 접근하는 경우가 발생하더라도 별 문제 없이 코드가 컴파일 및 실행이 됨. 단 CPU와 다른 내부 메모리 구조를 가지는 Xeon Phi 머신에서는, MPI를 이용해 코드를 컴파일 할 시 해당 부분들이 문제를 일으켜 코드가 정상적으로 작동하지 않음. Xeon Phi와 MPI 환경에서의 VASP 의 성능을 시험하기 위해서는 해당 부분들에서의 메모리 접근 오류들을 먼저 해결해야 할 것으로 보이지만 많은 시간이 필요할 것으로 예상

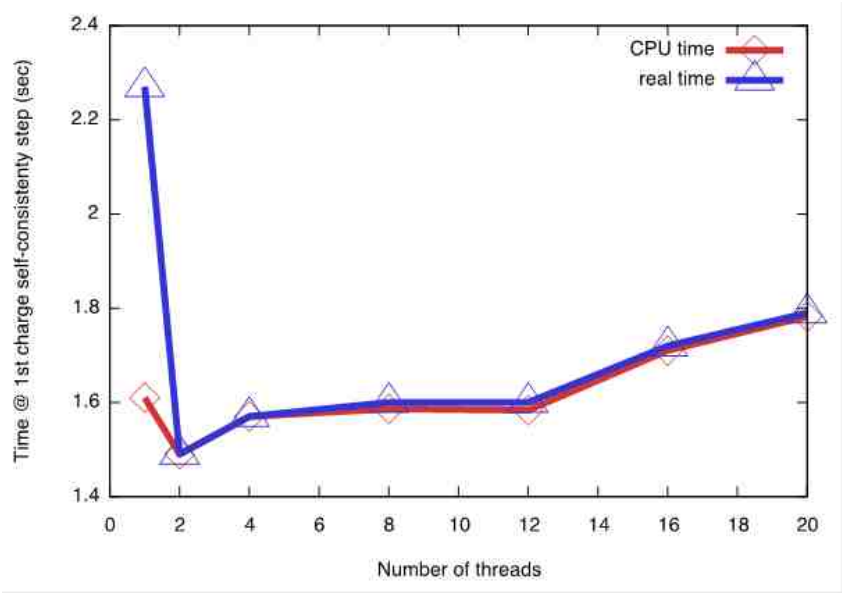
* Xeon Phi에서 Native 수행시 CPU보다 성능이 느리며 원자 수를 늘이면 조금 더 성능 향상은 기대해볼 수 있을 것으로 판단됨

- SrIrO3: CPU(2 Threads: 1.49초)대비 Xeon Phi(32 Threads: 24.35초)가 약 16배 성능저하

· 다음 그림은 SrIrO3을 CPU에서 실행한 결과.

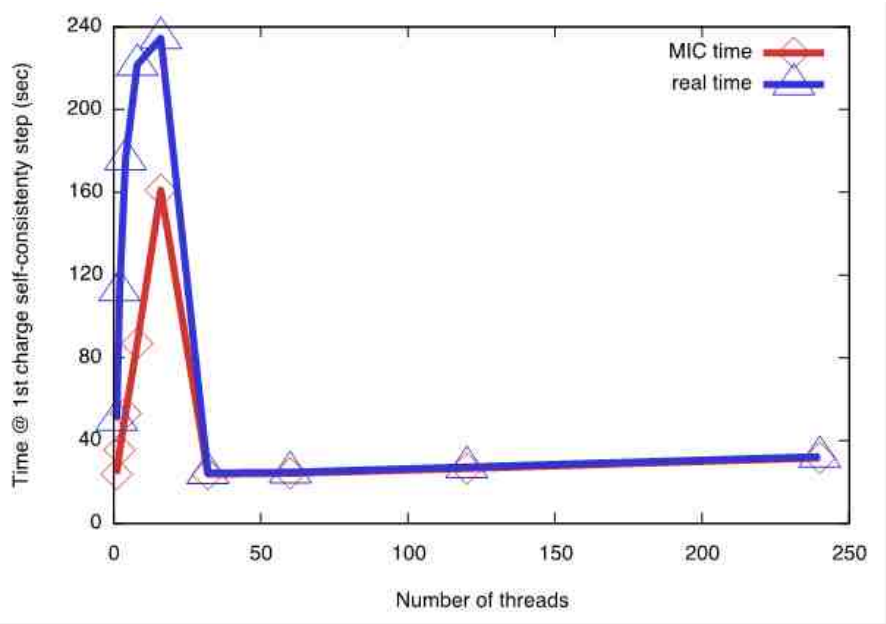
· 문제의 크기가 작기 때문에, thread 수가 2개일 때 오히려 가장 빠른 실행 속도를 보이며, thread 수가 그 이상일 때는 오히려 계산 분배에 따른 overhead때문에 계산 시간이 오히려 느려짐

· Thread 수가 1개일 때는 0.6초의 CPU 계산 시간과 실제 wall clock time 사이의 차이가 있는데, thread 수가 2개 이상이 되면서부터는 cpu time과 real time 사이의 차이가 거의 사라지는 것을 볼 수 있지만 이러한 현상에 대한 이유는 알기 힘듬



SrIrO3 CPU에서 계산할 때, thread 수 별 첫 charge self-consistency loop 실행에 걸리는 시간. 붉은색 선은 실제 계산이 CPU에서 실행된 시간, 그리고 푸른색 선은 wall clock time

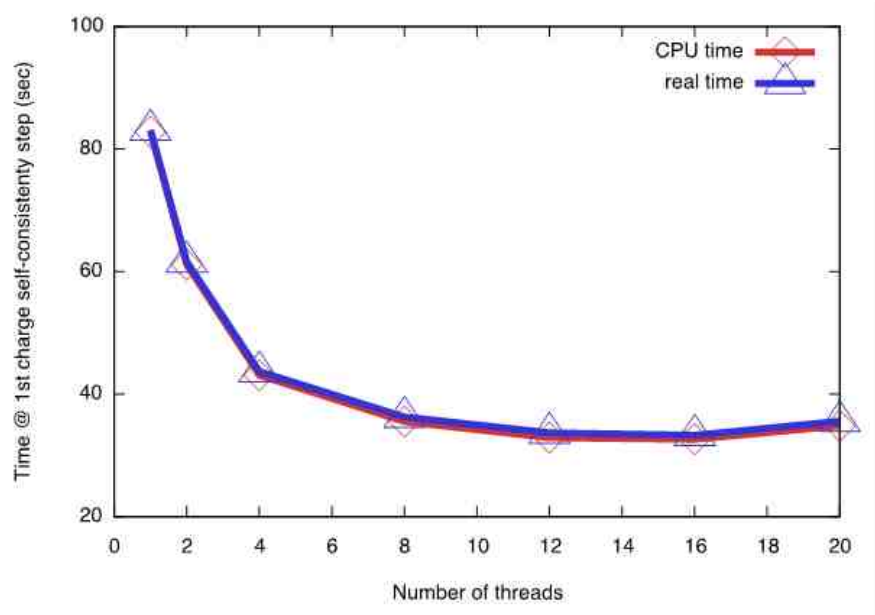
- 다음 그림은 Xeon Phi에서의 실행 결과
- 특이하게도, thread 수가 32개보다 적은 시점에서는 thread 수를 증가시키는 것이 오히려 전체적인 성능 저하를 유발하며, 성능 저하는 thread 수가 16개일 때 제일 심각하여, 순수 계산 시간의 경우 5~6배에 가까이 늘어남
- Thread 수가 32개 이상이 될 경우는 계산 속도가 개선이 되지만, thread가 1개일 때에 비해서 성능 향상에 대해 확인어려움



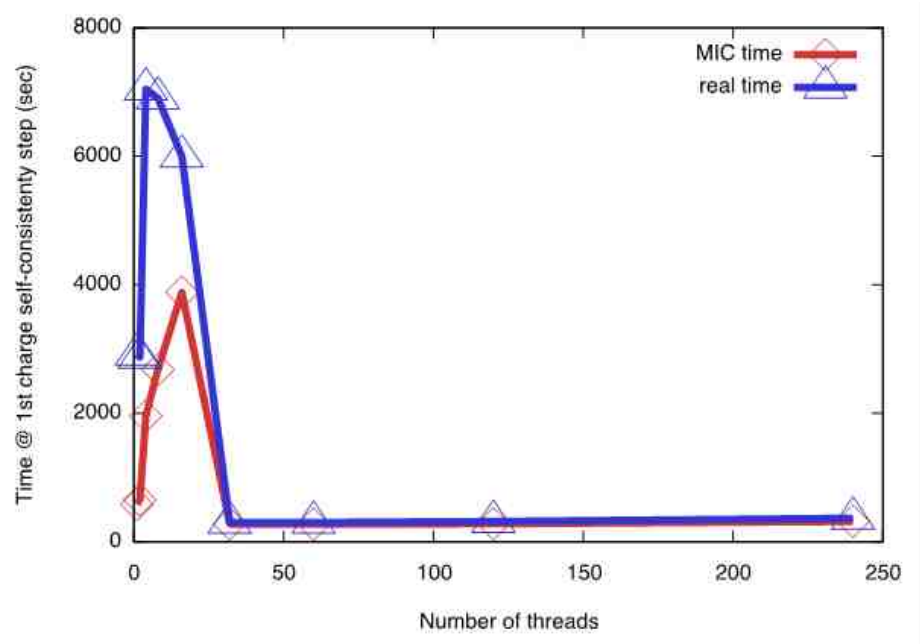
SrIrO3 Xeon Phi에서 계산할 때, thread 수 별 첫 charge self-consistency loop 실행에 걸리는 시간. 붉은색 선은 실제 계산이 Xeon Phi에서 실행된 시간, 그리고 푸른색 선은 wall clock time

· CPU에서의 실행 속도에 비해, Xeon Phi Native 모드의 실행 속도는, 양쪽의 실행 속도가 가장 빠를 때를 비교하면, CPU의 경우 1.49초 (2 threads), 그리고 Xeon Phi의 경우 24.35초 (32 threads)로 Xeon Phi가 대략 16배 느린 것을 알 수 있음
· 병렬화 효율을 크게 기대할 수 없는 작은 물질이기 때문에 더욱 CPU에 비해 성능의 이득이 없는 결과를 얻었다고 생각할 수 있지만, 그 점을 감안하더라도 CPU에 비해 성능 저하가 큼

- GaTa4Se8: CPU(16 Threads: 33.3초)대비 Xeon Phi(60 Threads: 300.1초)가 약 9배 성능저하
· 다음 그림은 GaTa4Se8에서의 CPU thread 수에 따른 계산 시간의 변화
· 대상 물질의 크기가 작았던 SrIrO3경우와는 달리, GaTa4Se8에서는 thread 수가 12개에 도달할 때 까지 병렬화의 이득을 확인할 수 있으며, 또한 CPU에서 계산이 실행되는 시간과 실제 wall clock time 사이의 차이를 확인할 수 없음.
· 비록 비교적 큰 물질에서도 thread 수에 따른 병렬화 효율의 linear scaling을 확인할 수는 없었지만 SrIrO3과 GaTa4Se8에서 보이는 경향을 통해 보다 큰 물리계에서는 개선된 thread 수에 따른 병렬화 효율을 얻을 수 있으리라고 기대할 수 있음



GaTa4Se8 CPU에서 계산할 때, thread 수 별 첫 charge self-consistency loop 실행에 걸리는 시간



GaTa4Se8을 Xeon Phi에서 계산할 때, thread 수 별 첫 charge self-consistency loop 실행에 걸리는 시간. 시간축으로 확대된 그림은 inset에서 볼 수 있음

- 위 그림에서 볼 수 있는 것처럼, Xeon Phi의 경우는 물질의 크기에 관계없이 유사한 현상(2~32 thread에서의 급격한 성능 저하)을 볼 수 있음
- SrIrO3에서도 유사한 현상이 나타났던 것으로 보아 이것은 Xeon Phi 카드의 자체의 특성으로 보이며, Xeon Phi에서 최적의 성능을 위해서는 30

개 이상의 thread를 사용해야 하는것으로 결론을 내릴 수 있음

· GaTa4Se8에서는 thread 수가 32개에서 60개로 증가할 때 역시 성능의 향상을 볼 수 있지만 그 차이는 아주 미미하며 (303초에서 300초로 감소), 120개 이상부터는 오히려 성능이 저하됨

. 마지막으로, 이 물질의 경우 CPU의 경우는 thread 수가 16개일 때 33.3초, 그리고 Xeon Phi의 경우는 thread 수가 60개일 때 300.1초의 계산 시간이 소모. Xeon Phi가 CPU에 비해 약 9배 가량 느려짐.

. SrIrO3과 GaTa4Se8의 결과를 비교해 보면, 보다 큰 물질에서 Xeon Phi의 성능이 개선되는 경향을 보인다고 말할 수 있지만, 이를 확인하기 위해서는 보다 큰 크기를 가지는 물질에서의 테스트 결과가 필요

※ VASP 코드 수정 없이 Xeon Phi 내에서 백터 성능을 거의 활용할 수 없는 것으로 판단됨

※ 각각의 물질에 대해서, 사용된 thread수 또는 사용된 CPU/Xeon Phi의 여부에 관계 없이 계산 값은 소수점 8째 자리 이전에서 모두 같았음

* Xeon Phi에서 AO 테스트는 현재 MKL의 BLAS 모듈 중 ZGEMM에 대해서 Offload 수행을 통한 성능개선을 기대했으나 수행되지 않았음 (VASP내의 ZGEMM 수행시 AO를 위한 행렬조건에 부합되지 않는 것으로 판단: MKL ZGEMM 함수가 AO로 수행되기 위해서는 행렬 크기가 2048이 넘어야 할뿐만 아니라 행렬 크기가 16의 배수가 되어야 함)

※ FFT 수행이 VASP에서 많은 계산을 차지하므로 MKL FFT 모듈의 AO가 지원되면 성능 향상 기대됨

<부록: Xeon Phi Native 모드 컴파일(OpenMP, MPI 사용하지 않음)을 위한 VASP makefile>

VASP Library를 위한 makefile :

```
.SUFFIXES: .inc .f .F
#-----------------------------------------------------------------------
--
# Makefile for Portland Group F90/HPF compiler
# the makefile was tested only under Linux on Intel platforms
# however it might work on other platforms as well
#
# this release of vasp.4.lib contains lapack v2.0
# this can be compiled with pgf90 compiler if the option -O1 is used
#
# Mind: one user reported that he had to copy preclib.F diolib.F
#  dlexlib.F and drdatab.F to the directory vasp.4.4, compile the files
#  there and link them directly  into vasp
#  for no obvious reason these files could not be linked from the library
#
#-----------------------------------------------------------------------
--

# C-preprocessor
CPP    = gcc -E -P -C $*.F >$*.f
FC= ifort

CFLAGS = -mmic -O
FFLAGS = -mmic -O1 -FI -m64
FREE   =  -FR

DOBJ =   preclib.o  timing_.o  derrf_.o  dclock_.o   diolib.o  dlexlib.o
drdatab.o


#-----------------------------------------------------------------------
--
# general rules
#-----------------------------------------------------------------------
--
```

```
libdmy.a: $(DOBJ) lapack_double.o linpack_double.o lapack_atlas.o
      -rm libdmy.a
      ar vq libdmy.a $(DOBJ)

# files which do not require autodouble
lapack_min.o: lapack_min.f
      $(FC) $(FFLAGS) $(NOFREE) -c lapack_min.f
lapack_double.o: lapack_double.f
      $(FC) $(FFLAGS) $(NOFREE) -c lapack_double.f
lapack_single.o: lapack_single.f
      $(FC) $(FFLAGS) $(NOFREE) -c lapack_single.f
lapack_atlas.o: lapack_atlas.f
      $(FC) $(FFLAGS) $(NOFREE) -c lapack_atlas.f
linpack_double.o: linpack_double.f
      $(FC) $(FFLAGS) $(NOFREE) -c linpack_double.f
linpack_single.o: linpack_single.f
      $(FC) $(FFLAGS) $(NOFREE) -c linpack_single.f

.c.o:
      $(CC) $(CFLAGS) -c $*.c
.F.o:
      $(CPP)
      $(FC) $(FFLAGS) $(FREE) $(INCS) -c $*.f
.F.f:
      $(CPP)
.f.o:
      $(FC) $(FFLAGS) $(FREE) $(INCS) -c $*.f
```

VASP main code를 위한 makefile :

```
.SUFFIXES: .inc .f .f90 .F
#-----------------------------------------------------------------------
--
# Makefile for Intel Fortran compiler for Pentium/Athlon/Opteron
# based systems
# we recommend this makefile for both Intel as well as AMD systems
# for AMD based systems appropriate BLAS (libgoto) and fftw libraries
are
# however mandatory (whereas they are optional for Intel platforms)
# For Athlon we recommend
```

```
#  ) to link against libgoto (and mkl as a backup for missing routines)
#  ) odd enough link in libfftw3xf_intel.a (fftw interface for mkl)
# feedback is greatly appreciated
#
# The makefile was tested only under Linux on Intel and AMD platforms
# the following compiler versions have been tested:
#  - ifc.7.1  works stable somewhat slow but reliably
#  - ifc.8.1  fails to compile the code properly
#  - ifc.9.1  recommended (both for 32 and 64 bit)
#  - ifc.10.1 partially recommended (both for 32 and 64 bit)
#           tested build 20080312 Package ID: l_fc_p_10.1.015
#           the gamma only mpi version can not be compiles
#           using ifc.10.1
#  - ifc.11.1 partially recommended (some problems with Gamma only and
intel fftw)
#           Build 20090630 Package ID: l_cprof_p_11.1.046
#  - ifort.12.1 strongly recommended (we use this to compile vasp)
#           Version 12.1.5.339 Build 20120612
#
# it might be required to change some of library path ways, since
# LINUX installations vary a lot
#
# Hence check ***ALL*** options in this makefile very carefully
#-----------------------------------------------------------------------
--
#
# BLAS must be installed on the machine
# there are several options:
# 1) very slow but works:
#   retrieve the lapackage from ftp.netlib.org
#   and compile the blas routines (BLAS/SRC directory)
#   please use g77 or f77 for the compilation. When I tried to
#   use pgf77 or pgf90 for BLAS, VASP hang up when calling
#   ZHEEV  (however this was with lapack 1.1 now I use lapack 2.0)
# 2) more desirable: get an optimized BLAS
#
# the two most reliable packages around are presently:
# 2a) Intels own optimised BLAS (PIII, P4, PD, PC2, Itanium)
#     http://developer.intel.com/software/products/mkl/
#   this is really excellent, if you use Intel CPU's
#
```

```
# 2b) probably fastest SSE2 (4 GFlops on P4, 2.53 GHz, 16 GFlops PD,
#    around 30 GFlops on Quad core)
#  Kazushige Goto's BLAS
#  http://www.cs.utexas.edu/users/kgoto/signup_first.html
#  http://www.tacc.utexas.edu/resources/software/
#
#-----------------------------------------------------------------------
--

# all CPP processed fortran files have the extension .f90
SUFFIX=.f90

#=======================================================================
==
# MPI section, uncomment the following lines until
#   general  rules and compile lines
# presently we recommend OPENMPI, since it seems to offer better
# performance than lam or mpich
#
# !!! Please do not send me any queries on how to install MPI, I will
# certainly not answer them !!!!
#=======================================================================
==
#-----------------------------------------------------------------------
--
# fortran linker for mpi
#-----------------------------------------------------------------------
--

FC=ifort -openmp -mmic
FCL=$(FC)

#-----------------------------------------------------------------------
--
# additional options for CPP in parallel version (see also above):
# NGZhalf           charge density   reduced in Z direction
# wNGZhalf          gamma point only reduced in Z direction
# scaLAPACK         use scaLAPACK (recommended if mkl is available)
# avoidalloc        avoid ALLOCATE if possible
# PGF90             work around some for some PGF90 / IFC bugs
# CACHE_SIZE          1000 for PII,PIII, 5000 for Athlon, 8000-12000 P4,
```

```
PD
# RPROMU_DGEMV       use DGEMV instead of DGEMM in RPRO (depends on used
BLAS)
# RACCMU_DGEMV       use DGEMV instead of DGEMM in RACC (depends on used
BLAS)
# tbdyn              MD package of Tomas  Bucko
#-----------------------------------------------------------------------
--
#-----------------------------------------------------------------------
--
# whereis CPP ?? (I need CPP, can't use gcc with proper options)
# that's the location of gcc for SUSE 5.3
#
#  CPP_   =  /usr/lib/gcc-lib/i486-linux/2.7.2/cpp -P -C
#
# that's probably the right line for some Red Hat distribution:
#
#  CPP_   =  /usr/lib/gcc-lib/i386-redhat-linux/2.7.2.3/cpp -P -C
#
#  SUSE X.X, maybe some Red Hat distributions:

CPP_  =    ./preprocess  <$*.F  |  /usr/bin/cpp  -P  -C  -traditional
>$*$(SUFFIX)

# this release should be fpp clean
# we now recommend fpp as preprocessor
# if this fails go back to cpp
#CPP_=fpp -f_com=no -free -w0 $*.F $*$(SUFFIX)


#-----------------------------------------------------------------------
--

CPP   = $(CPP_) -DHOST=\"LinuxIFC\" -DIFC \
    -DPGF90 -Dkind8 \
    -DCACHE_SIZE=16000 \
    -DNGXhalf

#    -DVASP2WANNIER90
#    -DPGF90 -Dkind8 -Davoidalloc \
#    -DCACHE_SIZE=16000 \
```

```
FFLAGS =  -FR -names lowercase -assume byterecl -check bounds
# ifc.9.1, ifc.10.1 recommended
#OFLAG=-O2 -ip
OFLAG=-O2 -ip -I$(MKLROOT)/include

OFLAG_HIGH = $(OFLAG)
OBJ_HIGH =
OBJ_NOOPT =
DEBUG  = -FR -O0
INLINE = $(OFLAG)

#-----------------------------------------------------------------------
--
# location of SCALAPACK
# if you do not use SCALAPACK simply leave this section commented out
#-----------------------------------------------------------------------
--

# usually simplest link in mkl scaLAPACK
#BLACS= -lmkl_blacs_openmpi_lp64
#SCA= $(MKL_PATH)/libmkl_scalapack_lp64.a $(BLACS)

#-----------------------------------------------------------------------
--
# libraries
#-----------------------------------------------------------------------
--

MKL_FFTW_PATH=$(MKLROOT)/interfaces/fftw3xf/
LAPACK=     -Wl,--start-group     $(MKLROOT)/lib/mic/libmkl_intel_lp64.a
$ ( M K L R O O T ) / l i b / m i c / l i b m k l _ c o r e . a
$(MKLROOT)/lib/mic/libmkl_intel_thread.a -Wl,--end-group -lpthread -lm
LIB     = ../vasp.5.lib_mic/libdmy.a \
      ../vasp.5.lib_mic/linpack_double.o \
      $(LAPACK)


#-----------------------------------------------------------------------
--
```

```
# parallel FFT
#-----------------------------------------------------------------------
--

# FFT: fftmpi.o with fft3dlib of Juergen Furthmueller
#FFT3D  = fftmpi.o fftmpi_map.o fft3dfurth.o fft3dlib.o

# alternatively: fftw.3.1.X is slighly faster and should be used if
available
#FFT3D          =   fftmpiw.o   fftmpi_map.o   fftw3d.o   fft3dlib.o
/opt/libs/fftw-3.1.2/lib/libfftw3.a

# you may also try to use the fftw wrapper to mkl (but the path might
vary a lot)
# it seems this is best for AMD based systems
#FFT3D       =  fftmpiw.o     fftmpi_map.o     fftw3d.o     fft3dlib.o
$(MKLROOT)/lib/mic/libfftw2xf_double_intel.a
FFT3D  = fftw3d.o  fft3dlib.o  $(MKLROOT)/lib/mic/libfftw3xf_intel.a
INCS = -I$(MKLROOT)/include/fftw -I$(MKLROOT)/include
#INCS = -I/home/heungsikim/lib/

#-----------------------------------------------------------------------
--
# general rules and compile lines
#-----------------------------------------------------------------------
--
BASIC=  symmetry.o symlib.o  lattlib.o  random.o


SOURCE=  base.o    mpi.o     smart_allocate.o     xml.o  \
       constant.o jacobi.o  main_mpi.o  scala.o  \
       asa.o     lattice.o  poscar.o  ini.o  mgrid.o  xclib.o  vdw_nl.o
xclib_grad.o \
       radial.o  pseudo.o  gridq.o    ebs.o  \
       mkpoints.o wave.o    wave_mpi.o  wave_high.o  spinsym.o \
       $(BASIC)      nonl.o         nonlr.o       nonl_high.o  dfast.o
choleski2.o \
       mix.o     hamil.o   xcgrad.o  xcspin.o   potex1.o  potex2.o  \
       constrmag.o cl_shift.o relativistic.o LDApU.o \
       paw_base.o metagga.o  egrad.o    pawsym.o   pawfock.o  pawlhf.o
rhfatm.o  hyperfine.o paw.o  \
```

```
        mkpoints_full.o       charge.o   Lebedev-Laikov.o  stockholder.o
dipol.o   pot.o \
        dos.o     elf.o     tet.o     tetweight.o hamil_rot.o \
        chain.o   dyna.o    k-proj.o   sphpro.o   us.o  core_rel.o \
        aedens.o  wavpre.o  wavpre_noio.o broyden.o \
        dynbr.o   hamil_high.o  rmm-diis.o reader.o   writer.o   tutor.o
xml_writer.o \
        brent.o   stufak.o  fileio.o   opergrid.o stepver.o  \
        chgloc.o            fast_aug.o    fock_multipole.o        fock.o
mkpoints_change.o sym_grad.o \
        mymath.o        internals.o   npt_dynamics.o        dynconstr.o
dimer_heyden.o dvvtrajectory.o vdwforcefield.o \
        nmr.o     pead.o    subrot.o   subrot_scf.o \
        force.o   pwlhf.o   gw_model.o optreal.o  steep.o   davidson.o
david_inner.o \
        electron.o  rot.o     electron_all.o   shm.o          pardens.o
paircorrection.o \
        optics.o   constr_cell_relax.o   stm.o    finite_diff.o elpol.o
 \
        hamil_lr.o rmm-diis_lr.o  subrot_cluster.o subrot_lr.o \
        lr_helper.o hamil_lrf.o   elinear_response.o ilinear_response.o
\
        linear_optics.o \
        setlocalpp.o       wannier.o   electron_OEP.o   electron_lhf.o
twoelectron4o.o \
        mlwf.o        ratpol.o  screened_2e.o  wave_cacher.o  chi_base.o
wpot.o \
        local_field.o ump2.o ump2kpar.o fcidump.o ump2no.o \
        bse_te.o bse.o acfdt.o chi.o sydmat.o dmft.o \
        rmm-diis_mlr.o       linear_response_NMR.o   wannier_interpol.o
linear_response.o

vasp: $(SOURCE) $(FFT3D) $(INC) main.o
      rm -f vasp
      $(FCL) -o vasp main.o  $(SOURCE)   $(FFT3D) $(LIB) $(LINK)
makeparam: $(SOURCE) $(FFT3D) makeparam.o main.F $(INC)
      $(FCL) -o makeparam  $(LINK) makeparam.o $(SOURCE) $(FFT3D) $(LIB)
zgemmtest: zgemmtest.o base.o random.o $(INC)
      $(FCL) -o zgemmtest $(LINK) zgemmtest.o random.o base.o $(LIB)
dgemmtest: dgemmtest.o base.o random.o $(INC)
      $(FCL) -o dgemmtest $(LINK) dgemmtest.o random.o base.o $(LIB)
```

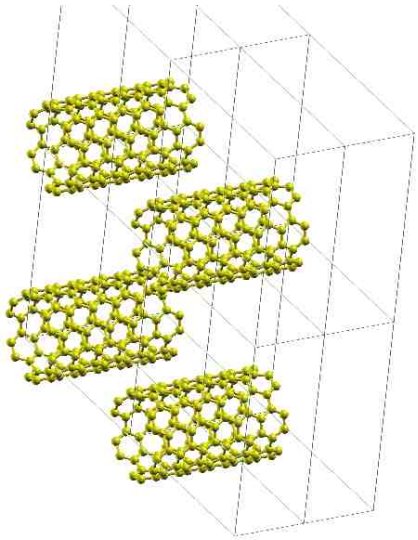
```
ffttest:  base.o  smart_allocate.o  mpi.o  mgrid.o  random.o  ffttest.o
$(FFT3D) $(INC)
      $(FCL)  -o  ffttest  $(LINK)  ffttest.o  mpi.o  mgrid.o  random.o
smart_allocate.o base.o $(FFT3D) $(LIB)
kpoints: $(SOURCE) $(FFT3D) makekpoints.o main.F $(INC)
      $(FCL) -o kpoints $(LINK) makekpoints.o $(SOURCE) $(FFT3D) $(LIB)

clean:
      -rm -f *.g *.f *.o *.L *.mod ; touch *.F

main.o: main$(SUFFIX)
      $(FC) $(FFLAGS) $(DEBUG)  $(INCS) -c main$(SUFFIX)
xcgrad.o: xcgrad$(SUFFIX)
      $(FC) $(FFLAGS) $(INLINE)  $(INCS) -c xcgrad$(SUFFIX)
xcspin.o: xcspin$(SUFFIX)
      $(FC) $(FFLAGS) $(INLINE)  $(INCS) -c xcspin$(SUFFIX)

makeparam.o: makeparam$(SUFFIX)
      $(FC) $(FFLAGS) $(DEBUG)  $(INCS) -c makeparam$(SUFFIX)

makeparam$(SUFFIX): makeparam.F main.F
#
# MIND: I do not have a full dependency list for the include
# and MODULES: here are only the minimal basic dependencies
# if one strucuture is changed then touch_dep must be called
# with the corresponding name of the structure
#
base.o: base.inc base.F
mgrid.o: mgrid.inc mgrid.F
constant.o: constant.inc constant.F
lattice.o: lattice.inc lattice.F
setex.o: setexm.inc setex.F
pseudo.o: pseudo.inc pseudo.F
mkpoints.o: mkpoints.inc mkpoints.F
wave.o: wave.F
nonl.o: nonl.inc nonl.F
nonlr.o: nonlr.inc nonlr.F

$(OBJ_HIGH):
      $(CPP)
      $(FC) $(FFLAGS) $(OFLAG_HIGH) $(INCS) -c $*$(SUFFIX)
```

```
$(OBJ_NOOPT):
      $(CPP)
      $(FC) $(FFLAGS) $(INCS) -c $*$(SUFFIX)

fft3dlib_f77.o: fft3dlib_f77.F
      $(CPP)
      $(F77) $(FFLAGS_F77) -c $*$(SUFFIX)

.F.o:
      $(CPP)
      $(FC) $(FFLAGS) $(OFLAG) $(INCS) -c $*$(SUFFIX)
.F$(SUFFIX):
      $(CPP)
$(SUFFIX).o:
      $(FC) $(FFLAGS) $(OFLAG) $(INCS) -c $*$(SUFFIX)

# special rules
#-----------------------------------------------------------------------
--
# these special rules have been tested for ifc.11 and ifc.12 only

fft3dlib.o : fft3dlib.F
      $(CPP)
      $(FC) -FR -lowercase -O2 -c $*$(SUFFIX)
fft3dfurth.o : fft3dfurth.F
      $(CPP)
      $(FC) -FR -lowercase -O1 -c $*$(SUFFIX)
fftw3d.o : fftw3d.F
      $(CPP)
      $(FC) -FR -lowercase -O1 $(INCS) -c $*$(SUFFIX)
fftmpi.o : fftmpi.F
      $(CPP)
      $(FC) -FR -lowercase -O1 -c $*$(SUFFIX)
fftmpiw.o : fftmpiw.F
      $(CPP)
      $(FC) -FR -lowercase -O1 $(INCS) -c $*$(SUFFIX)
wave_high.o : wave_high.F
      $(CPP)
      $(FC) -FR -lowercase -O1 -c $*$(SUFFIX)
# the following rules are probably no longer required (-O3 seems to
work)
```

```
wave.o : wave.F
      $(CPP)
      $(FC) -FR -lowercase -O2 -c $*$(SUFFIX)
paw.o : paw.F
      $(CPP)
      $(FC) -FR -lowercase -O2 -c $*$(SUFFIX)
cl_shift.o : cl_shift.F
      $(CPP)
      $(FC) -FR -lowercase -O2 -c $*$(SUFFIX)
us.o : us.F
      $(CPP)
      $(FC) -FR -lowercase -O2 -c $*$(SUFFIX)
LDApU.o : LDApU.F
      $(CPP)
      $(FC) -FR -lowercase -O2 -c $*$(SUFFIX)
```

2. Quantum Espresso

- 개요
  - 밀도 범함수(Density Functional Theory)를 기반으로 한 제일원리 계산을 수행하는 코드로서 분자 또는 결정 구조의 총 에너지를 구하고 전자구조 등을 계산. 또한 에너지를 최소화 하여 안정화된 구조를 얻을 수 있고 양자효과를 적용한 분자 동역학 계산을 수행가능
  - FFT(퓨리에) 변환과 행렬 연산이 핫스팟. BLAS, LAPACK, FFTW 등의 라이브러리를 사용하고 있으며, 해당 라이브러리의 성능이 중요
  - NVIDIA GPU는 CUDA로 개발되어 있으며, CUDA 기반 수치라이브러리 MAGMA과 CPU와 GPU의 작업 분배를 위한 라이브러리 Phigemm을 사용함
  - Xeon Phi는 Offload 방법은 개발중이며 Native 및 AO 방법을 통해 사용가능
- 수행 내용
  - 탄소 나노튜브(6,6)의 SCF(Self Consistent Field) 1회 계산수행 (단위 셀은 72개의 탄소 원자와 288개의 전자(core electron 제외 - core electron은 pseudo potential 방법 적용)를 포함)
  - k-point는 튜브 축 방향으로 8개로 설정하고 평면 파동함수의 kinetic energy cutoff 는 50 Ry를 사용한다.



(6,6) Unit-cell 구조

  - NVIDIA GPU는 CUDA를, Xeon Phi는 Native 및 AO(Automatic Offlod)를 이용해서 수행
    * 프로그램자체의 쓰레드 병렬화가 MPI 병렬화보다 성능이 좋지않은 특성 때문에 NVIDIA GPU카드 사용시 MPI 병렬화로 수행하여 함

• 설치 및 실행 방법

- Xeon Phi 설치 방법

1. Unpack the Quantum Espresso source code

```
tar -xzvf espresso-5.0.2.tar.gz
```

2. Source the Intel Compiler and MPI* software (adapt the version numbers to the version you have)

```
source /opt/intel/composer_xe_2013.2.146/bin/compilervars.sh intel64
source /opt/intel/impi/4.1.1/bin64/mpivars.sh
```

3. Create an initial configuration with defined Intel compilers:

```
export FC=mpiifort
export F90=$FC
export F77=$FC
export MPIF90=$FC
export FCFLAGS="-O3 -xAVX -fno-alias -ansi-alias -g -mkl -$MKLROOT/include/fftw"
export FFLAGS=$FCFLAGS
export CC=mpiicc
export CPP="icc -E"
export CFLAGS=$FCFLAGS
export AR=xiar
export BLAS_LIBS=""
export LAPACK_LIBS="-lmkl_blacs_intelmpi_lp64"
export FFT_LIBS="-L$MKLROOT/intel64"
./configure --enable-openmp --enable-parallel2. Source the Intel Compiler and MPI* software (adapt the version numbers to the version you have)
```

4. Edit the make.sys file in the top package directory and modify the DFLAGS variable in order to enable hybrid MPI+openmp mode, fftw3, and scalapack:

```
DFLAGS = -D__INTEL -D__FFTW3 -D__MPI -D__PARA -D__SCALAPACK -D__OPENMP
BLAS_LIBS = -lmkl_intel_thread -lmkl_core
LAPACK_LIBS = -lmkl_blacs_intelmpi_lp64
```

5. Build with the threaded Intel MKL, which is required to enable Intel MKL Automatic Offload.

```
make pw
```

- Xeon Phi 실행 방법

1. Set up the environment:

```
source /opt/intel/composer_xe_2013.2.146/bin/compilervars.sh intel64
source /opt/intel/impi/4.1.1/bin64/mpivars.sh
export MKLROOT=
export LD_LIBRARY_PATH=$MKLROOT/lib/intel64:$LD_LIBRARY_PATH
```

2. Use these additional settings to define the Intel MKL Automatic Offload thresholds:

```
export MKL_MIC_THRESHOLDS_ZGEMM=500,500,500
```

3. Set OMP_NUM_THREADS explicitly and PPN for pin script.

4. Enable the Intel MKL Automatic Offload feature with the following (also see ao_pin.sh example below):

```
MKL_MIC_ENABLE=1
```

5. Load the MPSS environment with the following shell script:

```
. /opt/intel/mpss/2.1.6720-16/etc/mpss_vars.sh
export MIC_SYSROOT=/opt/intel/mpss/2.1.6720-16
```

6. Start the binary with “mpirun” through the thread pinning proxy:

```
mpirun <mpi_options> ./ao_pin.sh ${path_to_pw}/pw.x <pw_options>
```

7. The pinning proxy requires correct thread placing on the Intel Xeon Phi coprocessor side. (부록 참고)

• 결과

- NVIDIA GPU 및 Xeon Phi 테스트

  * 단일노드에서 GPU 추가사용시 CPU(1core)대비 5.34배 성능향상, CPU(20cores)대비 1.94배 성능감소

  ※ Openmp 등의 thread 병렬화 성능이 MPI 병렬화 성능에 비해 매우 낮다는 특성 때문에 CPU의 모든 core를 다 활용하는 부분이 문제점으로 나타남

  ※ CPU(20cores)에 GPU 추가사용시 MPS를 동작시키면 성능향상을 기대할 수 있음

  * 단일노드에서 Xeon Phi Native 사용시 CPU(1core)대비 최대 4.44배 성능향상, CPU(20cores)대비 2.35배 성능감소

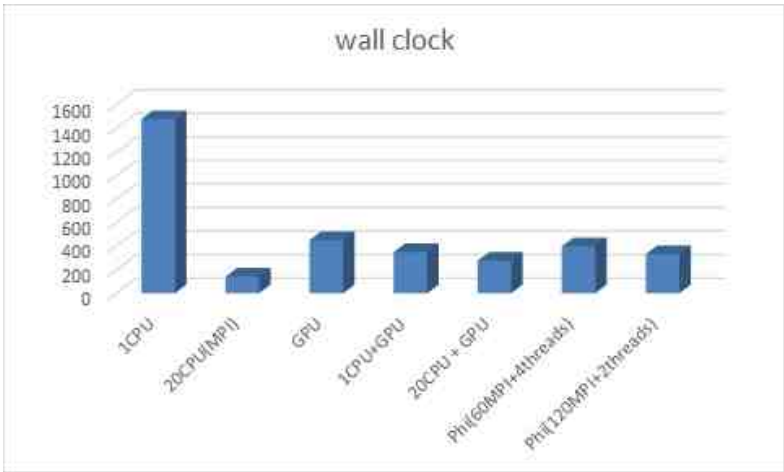
  ※ 파일 입출력이 모든 프로세스에서 수행되어 Native 사용시 성능 저하의 요인이 됨

  * 멀티노드에서도 단일노드와 유사한 결과를 보여줌

  * Xeon Phi AO 사용은 가능하지만 Intel 사이트에서 보여준 결과(Ausurf112 벤치마크, 1.11배 향상)만큼 성능향상을 보여주지 못함 => 성능향상이 없음

CPU대비 NVIDIA GPU와 Xeon Phi에서 QE 테스트

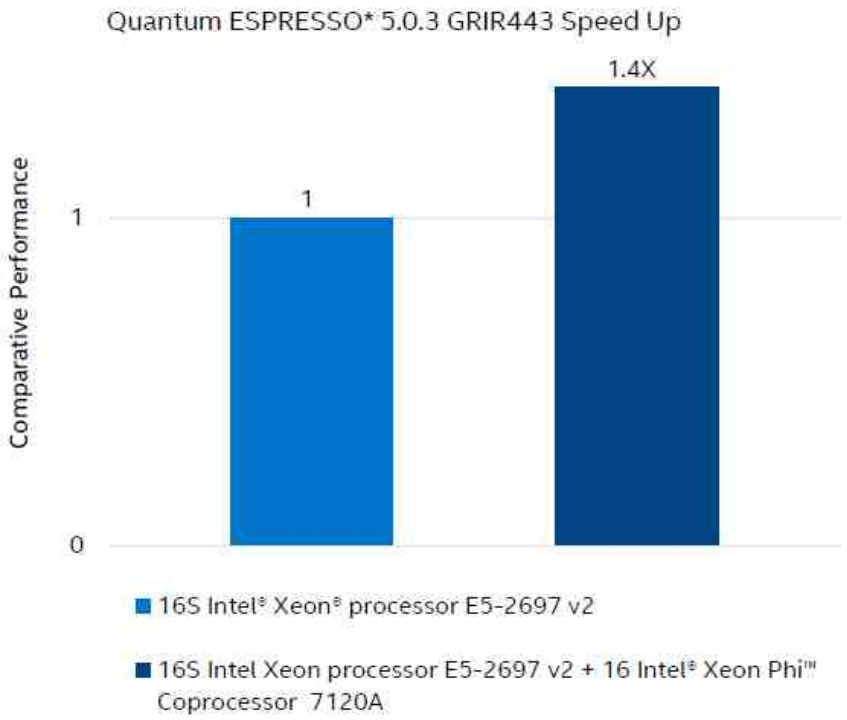
	wall clock (sec)
1 core	1478.75
20 cores(MPI)	141.55
GPU	450.76
1 core + GPU (MPS off)	349.99
20 cores + GPU (MPS off)	276.785
Xeon Phi(60cores + 4threads)	397.771
Xeon Phi(120cores + 2threads)	332.851



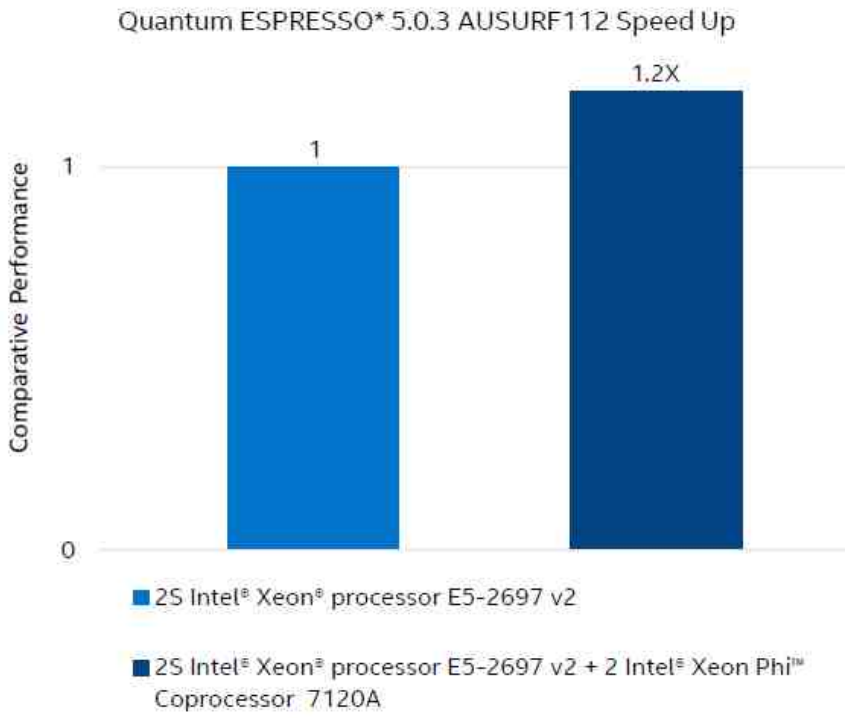
QE의 CPU, GPU, Xeon Phi 성능 비교

<부록: Intel 공식 테스트 결과>

* CPU 수행 대비 1.4배 향상 (GRIR443)
· 버전: Quantum ESPRESSO 5.0.3
· GRIR443은 멀티노드용 PRACE 벤치마크 Suite (http://qe-forge.org/gf/download/frsrelease/47/61/ GRIR443.tgz)
· 사용모델: OpenMP를 사용한 Offload, Intel MKL
· GRIR443은 워크로드가 큰 모델로서 규모가 있는 dense 행렬 곱셈이 많이 수행되고 Xeon Phi 7120에 이점을 많이 얻을 수 있음
· 결과: Xeon E5-2697-v2대비 1.4배 향상됨



* CPU 수행 대비 1.2배 향상 (AUSRUF112)
· 버전: Quantum ESPRESSO 5.0.3
· AUSRUF112은 단일노드용 DEISA 벤치마크 Suite (http://qe-forge.org/gf/download/frsrelease/45/78/ AUSURF112.tgz)
· 사용모델: OpenMP를 사용한 Offload, Intel MKL
· 결과: Xeon E5-2697-v2대비 1.2배 향상됨



* CPU 수행 대비 1.12배 향상 (AUSRUF112)

· 버전: Quantum ESPRESSO 5.0.2

· AUSRUF112은 단일노드용 DEISA 벤치마크 Suite (http://qe-forge.org/gf/download/frsrelease/45/78/ AUSURF112.tgz)

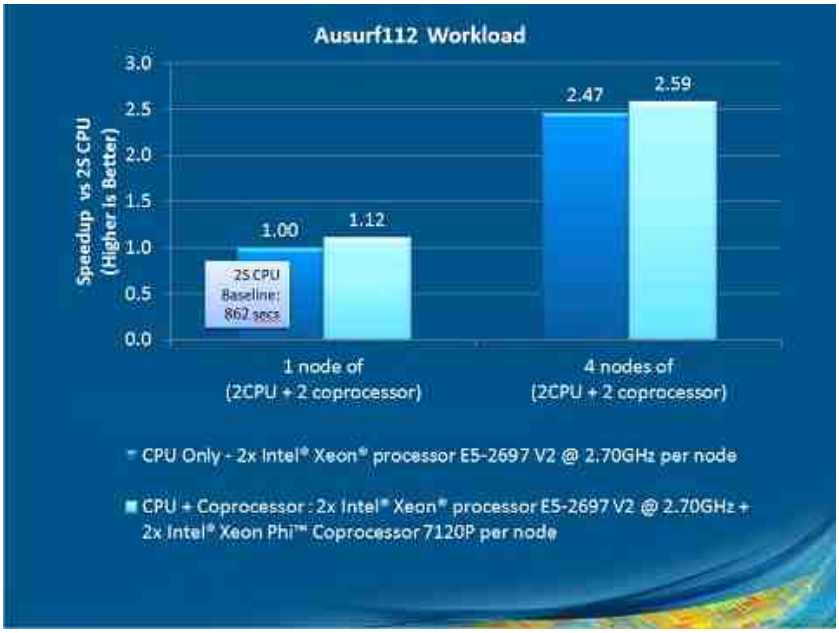
· 사용모델: Native 수행

· 노드를 1개와 4개에 대해서 각각 테스트, 노드당 2개의 CPU와 2개의 Xeon Phi카드 장착

CPU 수행	Xeon Phi 수행
1 node: MPI6, OMP4	1 node: MPI6, OMP4
4 nodes: MPI6, OMP4	4 nodes: MPI6, OMP4

※ CPU (MPI + OMP) - 4thread or 8thread - best chosen

※ Many Integrated Core - NUM Threads - 120thread or 240thread - best chosen



※ 서버 구성

2-socket/24 cores:

Processor: Intel® Xeon® processor E5-2697 V2 @ 2.70GHz (12 cores) with Intel® Hyper-Threading Technology5

Network: InfiniBand* Architecture Fourteen Data Rate (FDR)

Operating System: Red Hat Enterprise Linux* 2.6.32-358.el6.x86_64 #1 SMP Tue Jan 29 11:47:41 EST 2013 x86_64 x86_64 x86_64 GNU/Linux

Memory: 64GB

Coprocessor: 1X Intel Xeon Phi coprocessor 7120P: 61 cores @ 1.238 GHz, 4-way Intel Hyper-Threading Technology, Memory: 15872 MB

Intel® Many-core Platform Software Stack Version 2.1.6720-15

Intel® C++ Compiler Version 14.0

Intel® MPI Library Version 4.1

Intel® Math Kernel Library special engineering build

<부록: Shell Script 예제>

"ao_pin.sh"

```
#!/bin/sh
export MKL_MIC_ENABLE=1
export MKL_DYNAMIC=false
export MKL_MIC_DISABLE_HOST_FALLBACK=1

export MIC_LD_LIBRARY_PATH=$MKLROOT/lib/mic:$MIC_LD_LIBRARY_PATH

# Number of host cores
export HOST_CORE=$(( $PPN * $OMP_NUM_THREADS ))

# MPI process per node
MPI_PER_NODE=$PPN

# Number of MIC cores (minus 1)
MIC_CORE=60

export MKL_MIC_MAX_MEMORY=1G
export MIC_USE_2MB_BUFFERS=64K

MYNODE=$((PMI_RANK / MPI_PER_NODE))
MYRANK=$((PMI_RANK % MPI_PER_NODE))

HALFCORE=$((HOST_CORE / 2))

#export OMP_NUM_THREADS=$((HOST_CORE / MPI_PER_NODE))
export MIC_OMP_NUM_THREADS=$((MIC_CORE * 8 / MPI_PER_NODE))

# 2x oversubscribing on MIC
#export MIC_OMP_NUM_THREADS=$((MIC_CORE * 16 / MPI_PER_NODE))

HOST_FROM=$((MYRANK * OMP_NUM_THREADS))
HOST_TO=$(((MYRANK + 1) * OMP_NUM_THREADS - 1))

MHOST_FROM=$((MYRANK * MIC_OMP_NUM_THREADS + 1))
```

```
MHOST_TO=$(((MYRANK + 1) * MIC_OMP_NUM_THREADS))

if [ "$HOST_FROM" -lt "$HALFCORE" ]; then
export OFFLOAD_DEVICES=0
else
export OFFLOAD_DEVICES=1
MHOST_FROM=$((MHOST_FROM - MIC_CORE * 4))
MHOST_TO=$((MHOST_TO - MIC_CORE * 4))
fi

export  KMP_AFFINITY=explicit,granularity=fine,proclist=[${HOST_FROM}
        -${HOST_TO}:1]
export  MIC_KMP_AFFINITY=explicit,granularity=fine,proclist=[${MHOST_
        FROM}-${MHOST_TO}:1]

echo "[${PMI_RANK}] ${OFFLOAD_DEVICES}: ${OMP_NUM_THREADS} => ${KMP_A
        FFINITY}, ${MIC_OMP_NUM_THREADS} => ${MIC_KMP_AFFINITY}"

numactl --cpunodebind=${OFFLOAD_DEVICES} $@
```

"run.sh"

```
#!/bin/sh
export QE_ROOT=$PWD

cd $QE_ROOT
export WID=AUSURF112
export WDIR=$QE_ROOT/../workloads/$WID
export CORES=$(( $NODES * $PPN * $OMPN ))
export  RUNDIR=./rundir/${WID}_${CORES}c_${NODES}x${PPN}x${OMPN}_o_$L
        SB_JOBID
export BIN=$QE_ROOT/PW/src/pw.x

mkdir -p $RUNDIR
cd $RUNDIR
cp $WDIR/* .
echo "Stay in $PWD"

export OMP_NUM_THREADS=$OMPN
export MKL_NUM_THREADS=$OMPN

export I_MPI_FALLBACK_DEVICE=disable
export I_MPI_FABRICS=shm:dapl
export I_MPI_PIN=disable
export I_MPI_DEBUG=5

export MKL_MIC_ZGEMM_AA_M_MIN=500
export MKL_MIC_ZGEMM_AA_N_MIN=500
export MKL_MIC_ZGEMM_AA_K_MIN=500
export MKL_MIC_THRESHOLDS_ZGEMM=500,500,500


ldd $BIN > pw.ldd.out
mpirun -perhost $PPN -np $CPUS  $LS_SUBCWD/ao_pin.sh $BIN -in ./ausur
        f.in  2>&1 |tee ./run_${WID}_${CPUS}_${PPN}x${OMPN}omp.mklao.
        log
```

3. WRF

- 개요
  - WRF(Weather Research and Forecasting)는 대기과학 연구와 현업 예보를 위하여 만들어진 중규모 수치 예보이며 미국 기상연구소(NCAR: National Center for Atmospheric Research)에서 개발한 오픈소스 프로그램
  - 기존에 연구자들과 현업기관에서 널리 쓰이던 MM5모델을 대체하기 위한 차세대 수치모델로 개발되었으며, 지구를 3차원의 격차체계로 나누어 대기의 상태를 물리법칙에 따라 수식화한 모델
  - 역학코어와 여러 가지 물리과정을 포함하며 KISTI 4호기 슈퍼컴퓨터뿐만 아니라 전세계적으로 빈번하게 사용됨
  - WRF는 공식적으로 130여개의 나라에서 25,000여명이 이용하고 있고, 매년 workshop과 tutorial이 열리고 있음. 다음 웹사이트(wrf-model.org)에서 WRF에 대한 자세한 정보를 얻을 수 있음
  - CPU에서도 MPI 병렬화를 통해서 고해상도에 대해서 수행이 가능하지만, 계산시간이 많이 소요되는 물리과정에 대해서 계산가속기로 이식하여 전체 실행성능을 향상시킬 수 있음
  - NVIDIA GPU에서 물리과정 수행을 위해 CUDA와 OpenACC 언어 기반으로 비공개로 개발중
  - Xeon Phi는 Offload 방식으로 공식적으로 제공되지 않으며 Native/Symmetric 방식으로 수행방법에 대한 가이드가 지원됨
  - WRF를 수행하기 위해서는 초기 및 경계 자료가 필요하며, 이를 위하여 재분석장과 일일 해수면 온도 분석장을 이용하여 구축 할 수 있음
  - 초기 및 경계 자료를 확보 후, 이 자료를 모델 수행 영역에 맞게 변환하는 WPS 또는 WRF preprocess 라는 과정을 수행
  - 이 과정이 완료되면 real.exe를 실행하여 모델을 수행하기 전 초기화 과정을 완료하고, wrf.exe를 수행하여 수치 방정식을 적분하고 미래의 대기상태를 예측

• 수행 내용

- NVIDIA GPU 벤치마크
  * OpenACC 언어로 구현된 개발용 버전을 테스트 수행 (WRF 3.6)
  * CONUS 18km(80x80 격자)를 다음 역학 및 물리 설정으로 실험
- Xeon Phi 벤치마크
  * Native/Symmetric 방법으로 CONUS 12km, 2.5km에 대해서 다음 역학 및 물리 설정으로 WRF 3.6 실험

NVIDIA GPU Xeon Phi 테스트를 위한 역학 설정

역학 (Dynamics) Scheme
Fully compressible and nonhydrostatic model
Terrain-following hydrostatic pressure
Arakawa C-grid staggering
Runge-Kutta 2nd and 3rd order time integeration scheme
2nd and 6th order advection scheme

NVIDIA GPU와 Xeon Phi 테스트를 위한 물리 설정

물리(Physics) Scheme	NVIDIA GPU	Xeon Phi
Microphysics	Thompson	Morrison
Cumulus Parameterization	Kain-Fritsch	Kain-Fritsch
Longwave Radiation	RRTMG	RRTMG
Shortwave Radiation	Goddard	RRTMG
Surface Layer	MM5 similarity	Pleim
Land surface	Noah-MP	Pleim-Xiu
PBL	YSU	ACM2

· WRF는 비정역학계 모델로 terrain-following 기압 분포를 사용하고, Runge-Kutta 방법 등을 사용하여 역학과정에서 발생할 수 있는 수치적인 오차를 줄였음

· Xeon Phi 물리과정
  - 공기 중에 존재하는 수상체들을 표현하는 microphysics는 Morrison convection 과정을 표현하는 cumulus는 Kain-Fritsch
  - Longwave 및 shortwave Radiation을 표현하는 RRTMG
  - 대기와 지표간의 상호작용을 표현하는 Surface Layer과 Land surface으로는 Pleim과 Pleim-Xiu
  - 지표면 경계층 모수화로 ACM2을 사용

· NVIDIA GPU 물리과정

- 공기 중에 존재하는 수상체들을 표현하는 microphysics으로는 Thompson
- convection 과정을 표현하는 cumulus으로는 Kain-Fritsch
- Longwave 및 shortwave Radiation을 표현하는 RRTMG와 Goddard
- 대기와 지표간의 상호작용을 표현하는 Surface Layer와 Land surface으로는 MM5 similarity와 Noah-MP
- 지표면 경계층 모수화로 YSU를 사용

• 결과

- NVIDIA GPU 테스트

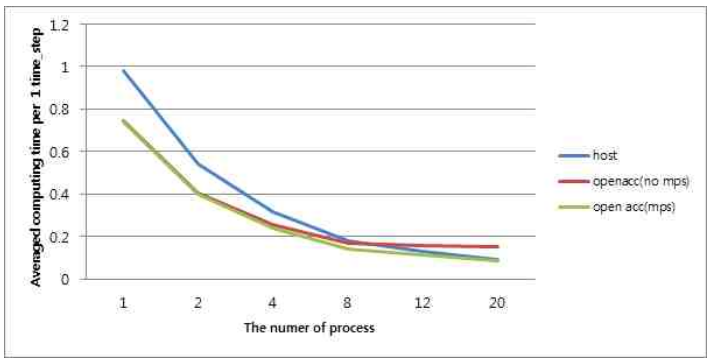
  * 단일노드에서 CPU(2cores)대비 GPU(MPS 동작, CPU(2cores)+1GPU) 사용시 성능 1.36배 향상

  - MPS를 작동시킬 경우에 코어수가 4개 이상인 경우에도 병렬확장성 보장이 됨

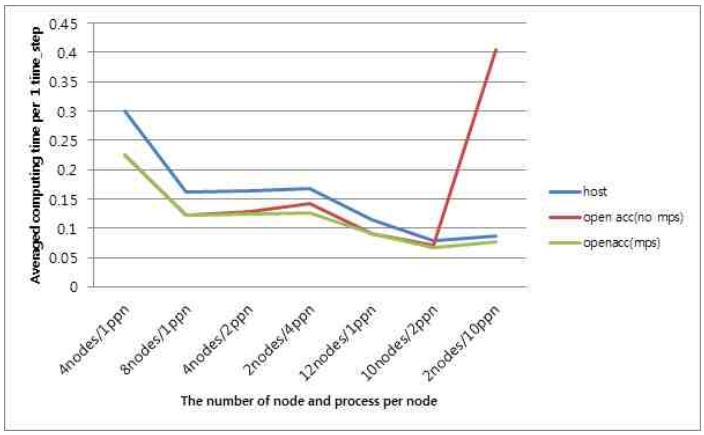
  * 멀티노드에서 CPU(4nodes/1ppn)대비 GPU(MPS 동작, CPU(4nodes/1ppn)+4GPUs) 사용시 성능 1.34배 향상

  - 8nodes/1ppn, 4nodes/2ppn, 2nodes/4ppn는 모두 8개의 프로세스를 MPI로 동작한 경우인데, MPS를 동작시키지 않으면 4ppn인 경우에 성능저하가 목격됨 -> 하지만 MPS를 동작시키면 모두 비슷한 성능이 보장됨을 알 수 있음

  - 10nodes/2ppn, 2nodes/10ppn는 모두 20개의 프로세스를 MPI로 동작한 경우인데, MPS를 동작시키지 않으면 10ppn인 경우에 성능저하가 현저하게 목격됨 -> 하지만 MPS를 동작시키면 모두 비슷한 성능이 보장됨을 알 수 있음



WRF의 GPU 성능

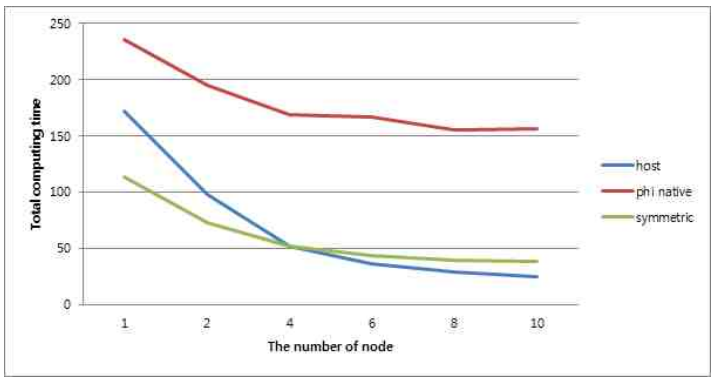


멀티노드에서 WRF의 GPU 성능

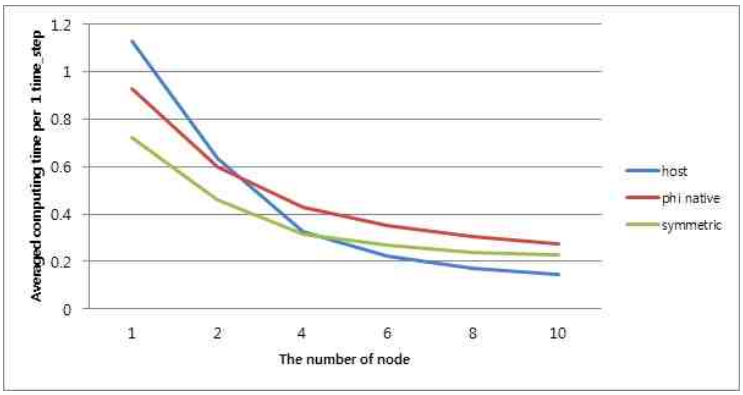
- Xeon Phi 테스트

  * CONUS 12km

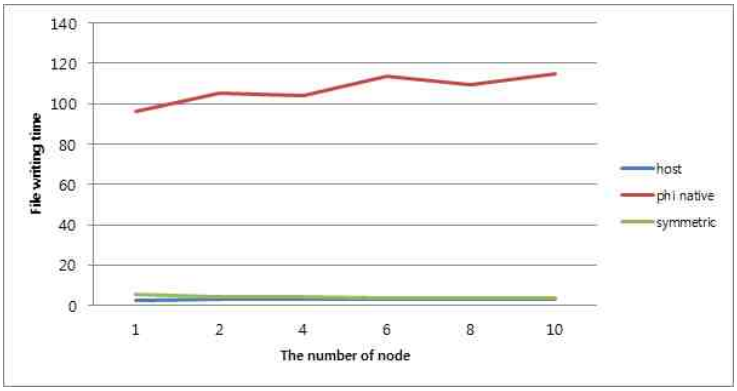
  - 단일노드 수행시 CPU(10ppn+2threads)대비 Native 수행시 1.27배 성능 감소, Symmetric 수행시 1.52배 성능향상
  - 멀티노드 수행시 CPU대비 Native 수행시 성능이 모두 감소, Symmetric 수행시 4노드부터 성능 감소
  - Native 수행시 계산자체는 2노드까지 CPU보다 성능이 낫지만 파일입출력으로 인해 성능저하
  - Symmetric 수행은 파일입출력을 담당하는 프로세스를 호스트에서만 수행하게 설정함으로서 파일 입출력으로 인한 성능저하가 없음



WRF Xeon Phi 전체 계산시간 비교
(Conus 12km)



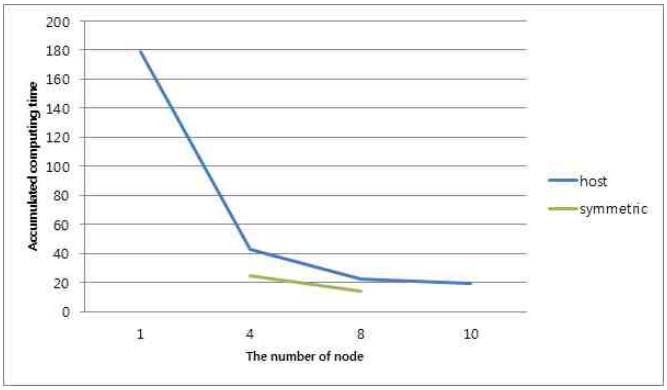
WRF의 Xeon Phi 계산시간 비교
(Conus 12km, 파일입출력 제외)



WRF Xeon Phi 파일입출력 시간 비교
(Conus 12km, 계산시간 제외)

* CONUS 2.5km

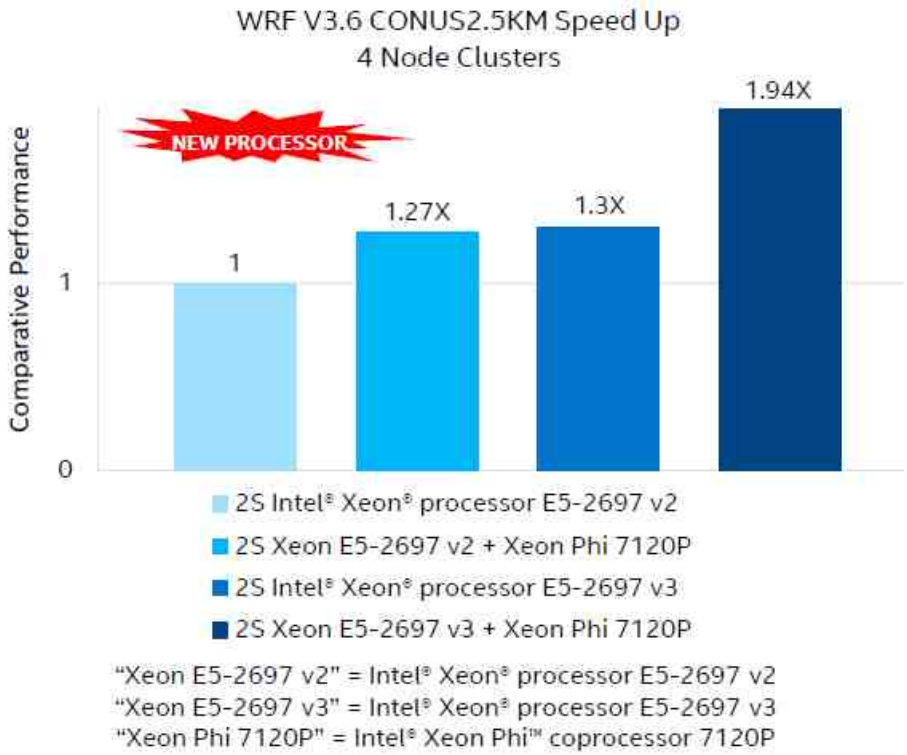
- CPU(10ppn+2threads)대비 Symmetric 수행시 4노드는 1.7배, 8노드는 1.59배 성능향상



WRF의 Xeon Phi 전체 계산시간 비교 (Conus 2.5km)

<부록: Intel 공식 테스트 결과>

* CPU 수행 대비 1.94배 향상 (Symmetric 수행)
· 버전: WRF 3.6
· 사용모델: Native, Symmetric
· 결과: Symmetric 사용시 Xeon E5-2697-v2대비 1.94배 향상됨



<부록: CPU에서 설치 및 실행방법>

```
$ module purge
$ module add intel_cluster_studio-2013_beta
$ vi ./bashrc
------------------- ./bashrc 에 다음 내용 추가
ulimit -s 262140
if [ `uname -m` == 'x86_64' ]
then
module purge
module add intel_cluster_studio-2013_beta
else
e x p o r t
PATH=${PATH}:/applic/compilers/intel/2013sp1/impi/4.1.3.048/mic/bin
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/applic/compilers/intel/2013s
p1/impi/4.1.3.048/mic/lib:/applic/compilers/intel/2013sp1/composer_xe_
2013_sp1.2.144/compiler/lib/mic:/home01/pop08/phi/netcdf/lib
fi
-------------------
$ source ~/.bashrc
$ which mpirun
다음 경로가 출력되는지 확인
/applic/compilers/intel/2013sp1/impi/4.1.3.048/intel64/bin/mpirun
```

COMPILE ZLIB

```
$ mkdir /home01/pop08/xeon/zlib
$tar-xvfzlib-1.2.8.tar.gz
$cd./zlib-1.2.8
$ CC=icc CFLAGS='-O3 -xHost -ip' ./configure  --prefix=/home01/pop08/xeo
n/zlib
$ make
$makeinstall
```

COMPILE HDF5

```
$ mkdir /home01/pop08/xeon/hdf5
$tar-xvfhdf5-1.8.6.tar.gz
$cd./hdf5-1.8.6
$ CC=icc F9X=ifort CXX=icpc ./configure --prefix=/home01/pop08/xeon/hdf
5 --enable-fortran --enable-cxx --with-zlib=/home01/pop08/xeon/zlib
$ make
$makeinstall
```

COMPILE NETCDF4

```
$ mkdir /home01/pop08/xeon/netcdf
$tar-xvfnetcdf-4.3.2.tar.gz
$tar-xvfnetcdf-fortran-4.4.0.tar.gz
$cd./netcdf-4.3.2
$ export CC=icc
$exportCXX=icpc
$exportCFLAGS='-O3-xHost-ip-no-prec-div-static-intel'
$exportCXXFLAGS='-O3-xHost-ip-no-prec-div-static-intel'
$exportF77=ifort
$exportFC=ifort
$exportF90=ifort
$exportFFLAGS='-O3-xHost-ip-no-prec-div-static-intel'
$exportCPP='icc-E'
$exportCXXCPP='icpc-E'
$exportLD_LIBRARY_PATH=/home01/pop08/xeon/hdf5/lib:$LD_LIBRARY_PATH
$ ./configure --prefix=/home01/pop08/xeon/netcdf LDFLAGS='-L/home01/pop
08/xeon/hdf5/lib -L/home01/pop08/xeon/zlib/lib' CPPFLAGS='-I/home01/pop
08/xeon/hdf5/include -I/home01/pop08/xeon/zlib/include' --disable-netcd
f-4
$ make
$makeinstall

$ cd ../netcdf-fortran-4.4.0
$ export LD_LIBRARY_PATH=/home01/pop08/xeon/netcdf/lib:$LD_LIBRARY_PATH
$./configure --prefix=/home01/pop08/xeon/netcdf LDFLAGS=-L/home01/pop08
/xeon/netcdf/lib CPPFLAGS=-I/home01/pop08/xeon/netcdf/include
$ make
$makeinstall
```

make 중 procdure 관련 error가 난다면 다음 파일에서 해당 procedure를 삭제한다 (line 8~23).

```
$vifortran/netcdf_overloads.f90
```

COMPILE WRF

```
$ tar -xvf WRFV3.6.TAR.gz
$cd./WRFV3
$ export NETCDF=/home01/pop08/xeon/netcdf
$ ./configure
```

Xeon(SNBwithAVXmods)

```
$ vi configure.wrf
```

mpicc를 mpiicc로 변경

mpifort를 mpiifort로 변경 또는 mpif90을 mpiifort로 변경
-O3를 -O2로 변경
$ vi dyn_em/module_advect_em.F
!DEC$vectoralways를 !DEC$ SIMD로 변경 (line 7578)
$ ./compile em_real >& log.log &
$ ls main/*.exe 로 확인

Configuring to run CONUS 12km

https://software.intel.com/en-us/articles/how-to-get-wrf-running-on-the-intelr-xeon-phitm-coprocessor

1. Download the CONUS12_rundir from http://www.mmm.ucar.edu/WG2bench/conus12km_data_v3/ and place it in ../WRFV3.

2. Copy the binary from ../WRFV3/main/wrf.exe to the ../CONUS12_rundir/wrf.exe

3. You need to copy/link at least all files in WRFV3/run to the directory where you are executing wrf.exe (i.e. the CONUS12_rundir)

4. Edit the namelist.input to add "use_baseparam_fr_nml = .t." under the &dynamics heading without the quotation marks (").

5. cd into the CONUS12_rundir and execute WRF with the runtime parameters in the below script.

Here is an example script to run it on an Intel® Xeon® host:

```
bash-4.1$ cat run.sh
source /opt/intel/composer_xe_2013.2.146/bin/compilervars.sh intel64
source /opt/intel/impi/4.1.0.030/intel64/bin/mpivars.sh
ulimit -s unlimited
I_MPI_PIN_MODE=mpd
OMP_NUM_THREADS=8
KMP_STACKSIZE=64m
KMP_AFFINITY=scatter,granularity=thread
KMP_BLOCKTIME=infinite
KMP_LIBRARY=turnaround
WRF_NUM_TILES=32
mpiexec.hydra -np 2 ./wrf.exe
```

4. The run is completed when it prints 'wrf: SUCCESS COMPLETE WRF, on the screen. You will have 2 files rsl.error.0000 and rsl.out.0000 in your CONUS12_rundir directory.

5. After the run, compute the total time taken to simulate 149 timesteps with the scripts below. The sum and mean values are of interest for WRF (lower the better).

The following script should help parse the output:

```
bash-4.1$ cat gettiming.sh
grep 'Timing for main' rsl.out.0000 | sed '1d' | head -149 | awk '{print
$9}' | awk -f stats.awk
bash-4.1$ cat stats.awk
BEGIN{ a = 0.0 ; i = 0 ; max = -999999999  ; min = 9999999999 }
{
   i ++
   a += $1
   if ( $1 > max ) max = $1
   if ( $1 < min ) min = $1
}
END{  printf("---n%10s    %8dn%10s    %15fn%10s    %15fn%10s    %15fn%10s
%        1        5        f        n        %        1        0        s
%15fn","items:",i,"max:",max,"min:",min,"sum:",a,"mean:",a/(i*1.0),"me
an/max:",(a/(i*1.0))/max) }
```

Configuring to run CONUS 2.5km

https://software.intel.com/en-us/articles/wrf-conus25km-on-intel-xeon-phi-coprocessors-and-intel-xeon-processors-in-symmetric-mode

1. Download the CONUS2.5_rundir from http://www2.mmm.ucar.edu/WG2bench/conus_2.5_v3/

2. Follow the READ-ME.txt to build the wrf input files.

3. The namelist.input has to be altered. The changes are as follows:
In the &time_interval section, edit the values as below:
restart_interval =360,
io_form_history =2,
io_form_restart =2,
io_form_input =2,
io_form_boundary =2,
Remove "perturb_input =.true." from the &domains section and replace with "nproc_x =8,"
Add "tile_strategy =2," under the &domains section.
Add "use_baseparam_fr_nml =.true." under the &dynamics section.

4. Create a new directory called CONUS2.5_rundir (../WRFV/CONUS_rundir) in the CONUS2.5_rundir, create 2 directories "mic" and "x86". Copy over contents of ../WRFV/run/ into “mic” and “x86” directories.

5. Copy the Intel Xeon Phi coprocessor binary into the CONUS2.5_rundir/mic directory and copy the Intel Xeon binary into the CONUS2.5_rundir/x64 directory.

6. Cd into the CONUS2.5_rundir and execute WRF as follows on 4 nodes (i.e 4 coprocessors + 4 Intel Xeon processors) in symmetric mode. To run conus2.5km, you need to have access to 4 nodes (example shown below)

RUN WRF

```
$ cd ./WRFV3/run

$ vi exec.sh
#!/bin/sh
ulimit-sunlimited
exportLD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/applic/compilers/intel/2013sp1/lib/intel64:/applic/compilers/intel/2013sp1/mkl/lib/intel64:/applic/compilers/intel/2013sp1/debugger/lib/intel64:/applic/compilers/intel/
```

2013sp1/composerxe/ipp/lib/intel64:/applic/compilers/intel/2013sp1/impi/4.1.3.048/intel64/lib:/applic/compilers/intel/2013sp1/vtune_amplifier_xe_2013/lib64:/applic/compilers/intel/2013sp1/advisor_xe_2013/lib64:/applic/compilers/intel/2013sp1/inspector_xe_2013/lib64:/applic/compilers/intel/2013sp1/itac/8.1.4.045/intel64/lib:/home01/pop08/xeon/netcdf/lib:/home01/pop08/xeon/hdf5/lib:/home01/pop08/xeon/zlib/lib
exportOMP_NUM_THREADS=2
exportKMP_STACKSIZE=32M
exportOMP_SCHEDULE=dynamic
exportKMP_AFFINITY=compact

$ mpiexec.hydra -host phi12 -n 10 ./exec.sh ./wrf.exe : -host phi04 -n 10 ./exec.sh ./real.exe
$mpiexec.hydra-hostphi12-n10./exec.sh./wrf.exe:-hostphi04-n10./exec.sh ./wrf.exe

Validation

To validate if the successful WRF run is correct or not, check the following:
It should generate a wrf_output file.
diffwrf your_output wrfout_reference >diffout_tag
'DIGITS' column should have high value (>3). If yes, the WRF run is considered valid.

Compiler Options

-mmic : build an application that natively runs on Intel® Xeon Phi™ Coprocessor
-openmp : enable the compiler to generate multi-threaded code based on the OpenMP* directives (same as -fopenmp)
-O3 :enable aggressive optimizations by the compiler.
-opt-streaming-stores always : generate streaming stores
-fimf-precision=low : low precision for higher performance
-fimf-domain-exclusion=15 : gives lowest precision sequences for Single precision and Double precision.
-opt-streaming-cache-evict=0 : turn off all cache line evicts.

<부록: Xeon Phi Native에서 설치 및 실행방법>

```
$ module purge
$moduleaddintel_cluster_studio-2013_beta
$ source /applic/compilers/intel/2013sp1/composer_xe_2013_sp1/bin/compilervars.sh intel64
$exportLD_LIBRARY_PATH=:${LD_LIBRARY_PATH}:/applic/compilers/intel/2013sp1/impi/4.1.3.048/mic/lib:/applic/compilers/intel/2013sp1/composer_xe_2013_sp1.2.144/compiler/lib/mic
$exportMIC_LD_LIBRARY_PATH=:${MIC_LD_LIBRARY_PATH}:/applic/compilers/intel/2013sp1/impi/4.1.3.048/mic/lib:/applic/compilers/intel/2013sp1/composer_xe_2013_sp1.2.144/compiler/lib/mic
```

COMPILE ZLIB

```
$ mkdir /home01/pop08/phi/zlib
$tar-xvfzlib-1.2.8.tar.gz
$cd./phi/zlib-1.2.8
$ CFLAGS=-mmic FCLAGS=-mmic LDFLAGS=-mmic ./configure --prefix=/home01/pop08/phi/zlib
$ make
$makeinstall
```

COMPILE hdf5

```
$ mkdir /home01/pop08/phi/hdf5
$tar-xvfhdf5-1.8.6.tar.gz
$cd./hdf5-1.8.6
```

다음 웹페이지를 참고하여 code를 수정 : [http://linux4hippos.blogspot.kr/2013/05/cross-compile-hdf5-for-intel-xeon-phi.html]

```
$ ./configure --prefix=/home01/pop08/phi/hdf5 --host=x86_64-k1om-linux CC=icc CXX=icpc FC=ifort CFLAGS="-mmic" CXXFLAGS="-mmic" FCFLAGS="-mmic" --with-zlib=/home01/pop08/phi/zlib --enable-fortran --enable-cxx
$ make
```

make 중 Error가 발생하면 참고하여 해결한다 (${host_hdf5}=xeon-host용 hdf5-1.8.13 디렉토리 위치. 즉 (${host_hdf5}=/home01/pop08/xeon/hdf5-1.8.13)

```
$ cp ${host_hdf5}/src/H5lib_settings.c ./src/
$make
```

```
$ cp ${host_hdf5}/src/H5Tinit.c ./src/
$make

$ vi test/links.c
:%s/HDmkdir/mkdir/g
$make
$ vi test/links_env.c
:%s/HDmkdir/mkdir/g
$make
$ vi perform/sio_engine.c
:%s/HDmkdir/mkdir/g
$make
$ cp ${host_hdf5}/fortran/src/H5fortran_detect.f90 ./fortran/src/
$make
$ cp ${host_hdf5}/fortran/src/H5fort_type_defines.h ./fortran/src/
$make
$ cp ${host_hdf5}/fortran/src/H5f90i_gen.h ./fortran/src/
$make
$ cp ${host_hdf5}/fortran/src/H5fortran_types.f90 ./fortran/src/
$make

$ make install
```

COMPILE CURL

```
$ CC=icc CXX=icc CFLAGS="-mmic" CXXFLAGS="-mmic" LDFLAGS="-mmic"
./configure --prefix=/home01/pop08/phi/zlib --host=x86_64-k1om-linux
--with-zlib=/home01/pop08/phi/zlib
make
make install
```

COMPILE NETCDF4

```
$ mkdir /home01/pop08/phi/netcdf
$tar-xvfnetcdf-4.3.2.tar.gz
$tar-xvfnetcdf-fortran-4.4.0.tar.gz
$cd./netcdf-4.3.2
$ export LD_LIBRARY_PATH=/home01/pop08/phi/hdf5/lib:$LD_LIBRARY_PATH
$exportMIC_LD_LIBRARY_PATH=/home01/pop08/phi/hdf5/lib:$MIC_LD_LIBRARY_
PATH

$ vi ./configure
as_fn_exit$as_status앞에 # 추가
```

```
$ CPPFLAGS="-DpgiFortran" CXX=icpc CC=icc F77=ifort FC=ifort ./configur
e NM=nm --prefix=/home01/pop08/phi/netcdf --host=x86_64-k1om-linux --bu
ild=x86_64-unknown-linux LDFLAGS="-L/home01/pop08/phi/hdf5/lib -L/home0
1/pop08/phi/zlib/lib" CPPFLAGS="-I/home01/pop08/phi/hdf5/include -I/hom
e01/pop08/phi/zlib/include" --disable-dap --disable-netcdf-4
$ make CFLAGS=-mmic FCFLAGS=-mmic LDFLAGS=-mmic
$makeinstall

$ cd netcdf-fortran-4.4.0
$ export LD_LIBRARY_PATH=/home01/pop08/phi/netcdf/lib:$LD_LIBRARY_PATH
$exportMIC_LD_LIBRARY_PATH=/home01/pop08/phi/netcdf/lib:$MIC_LD_LIBRAR
Y_PATH
$ vi ./configure
as_fn_exit$as_status앞에 # 추가
$ ./configure NM=nm --prefix=/home01/pop08/phi/netcdf --host=x86_64-k1o
m-linux --build=x86_64-unknown-linux LDFLAGS=-L/home01/pop08/phi/netcdf
/lib CPPFLAGS=-I/home01/pop08/phi/netcdf/include
$ make CFLAGS=-mmic FCFLAGS=-mmic FFLAGS=-mmic LDFLAGS=-mmic
$makeinstall
make 중 procdure 관련 error가 난다면 다음 파일에서 해당 procedure를 삭제한다
(line 8~23).
$vifortran/netcdf_overloads.f90
```

COMPILE WRF

```
$ tar -xvf WRFV3.6.TAR.gz
$cd./WRFV3
$ export NETCDF=/home01/pop08/phi/netcdf
$ ./configure
XeonPhi(MICarchitecture)
$ vi ./configure.wrf
replacempiccwithmpiicc
replacempifortwithmpiifort
replace-O3with-O2

$ vi dyn_em/module_advect_em.F
replace!DEC$vectorwith!DEC$SIMD
mpicc를 mpiicc로 변경
mpifort를 mpiifort로 변경 또는 mpif90을 mpiifort로 변경
-O3를 -O2로 변경
$ vi dyn_em/module_advect_em.F
!DEC$vectoralways를 !DEC$ SIMD로 변경 (line 7578)
```

```
$ ./compile em_real >& log.log &
$ ls main/*.exe 로 확인
```

RUN WRF

```
$ cd ./WRFV3/run

$ vi exec.sh
#!/bin/sh
ulimit-cunlimited
ulimit-sunlimited
exportLD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/applic/compilers/intel/2013sp1/impi/4.1.3.048/mic/lib:/applic/compilers/intel/2013sp1/composer_xe_2013_sp1.2.144/compiler/lib/mic:/home01/pop08/phi/netcdf/lib:/home01/pop08/phi/netcdf/lib:/home01/pop08/phi/hdf5/lib:/home01/pop08/phi/zlib/lib
exportOMP_NUM_THREADS=30
exportKMP_LIBRARY=turnaround
exportKMP_BLOCKTIME=infinite
exportKMP_STACKSIZE=64M
exportKMP_AFFINITY="balanced,granularity=thread"
```

Native 실행방법

```
$mpiexec.hydra-hostphi12-mic0-n8./exec.sh./wrf.exe:-hostphi04-mic0-n8./exec.sh./real.exe
$mpiexec.hydra-hostphi12-mic0-n8./exec.sh./wrf.exe:-hostphi04-mic0-n8./exec.sh./wrf.exe
```

Symmetric 실행방법

Host 용 wrf.exe와 real.exe를 wrf_host.exe와 real_host.exe로 복사해온다.

```
$ vi exec_host.sh
#!/bin/sh
ulimit-sunlimited
exportLD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/applic/compilers/intel/2013sp1/lib/intel64:/applic/compilers/intel/2013sp1/mkl/lib/intel64:/applic/compilers/intel/2013sp1/debugger/lib/intel64:/applic/compilers/intel/2013sp1/composerxe/ipp/lib/intel64:/applic/compilers/intel/2013sp1/impi/4.1.3.048/intel64/lib:/applic/compilers/intel/2013sp1/vtune_amplifier_xe_2013/lib64:/applic/compilers/intel/2013sp1/advisor_xe_2013/lib64:/applic/compilers/intel/2013sp1/inspector_xe_2013/lib64:/applic/compilers/intel/2013sp1/itac/8.1.4.045/intel64/lib:/home01/pop08/xeon/netcdf/lib:/home01/pop08/xeon/hdf5/lib:/home01/pop08/xeon/zlib/lib
```

```
exportOMP_NUM_THREADS=2
exportKMP_STACKSIZE=32M
exportOMP_SCHEDULE=dynamic
exportKMP_AFFINITY=compact

$ mpiexec.hydra -host phi12 -n 10 ./exec.sh ./real.exe : -host phi04 -n 10 ./exec.sh ./real.exe : -host phi12-mic0 -n 8 ./real_host.sh ./real_host.exe : -host phi04-mic0 -n 8 ./real_host.sh ./real_host.exe
$mpiexec.hydra-hostphi12-n10./exec.sh./wrf.exe:-hostphi04-n10./exec.sh ./wrf.exe:-hostphi12-mic0-n8./exec_host.sh./wrf_host.exe:-hostphi04-mi c0-n8./exec_host.sh./wrf_host.exe
```

<부록: NVIDIA GPU에서 OpenACC로 설치 및 실행방법>

OPENACC

READY

```
$ module purge
```

COMPILE ZLIB

```
$ CC=pgcc CFLAGS='-acc -Minfo=accel' ./configure --prefix=/home01/pop08/openacc/zlib
$ make
$makeinstall
```

COMPILE HDF5

```
$ CC=pgcc FC=pgf90 F90=pgf90 CXX=pgc++ CPPFLAGS="-DNDEBUG -DpgiFortran" CFLAGS="-O2 -Msignextend" FFLAGS="-O2 -w -V" ./configure --with-zlib=/home01/pop08/openacc/zlib --enable-fortran --prefix=/home01/pop08/openacc/hdf5
$ make
$makeinstall
```

COMPILE CURL

```
$ CC=pgcc CFLAGS="-O2 -Msignextend" ./configure --prefix=/home01/pop08/openacc/zlib --with-zlib=/home01/pop08/openacc/zlib
$ make
$makeinstall
```

COMPILE NETCDF

```
$ export LD_LIBRARY_PATH=/home01/pop08/openacc/hdf5/lib/:$LD_LIBRARY_PATH
$ CXX=pgc++ CC=pgcc F77=pgf77 FC=pgf90 ./configure --prefix=/home01/pop08/openacc/netcdf LDFLAGS="-L/home01/pop08/openacc/hdf5/lib -L/home01/pop08/openacc/zlib/lib" CPPFLAGS="-I/home01/pop08/openacc/hdf5/include -I/home01/pop08/openacc/zlib/include"
$ make
$makeinstall

$ cd netcdf-fortran
$ export LD_LIBRARY_PATH=/home01/pop08/openacc/netcdf/lib:$LD_LIBRARY_PATH
$ ./configure --prefix=/home01/pop08/openacc/netcdf LDFLAGS="-L/home01/pop08/openacc/netcdf/lib" CPPFLAGS="-I/home01/pop08/openacc/netcdf/incl
```

ude"
$ make
$makeinstall

COMPILE WRF
$ module purge
$source/applic/internal/testbed/pgi/linux86-64/14.7/openmpi.sh

$ export NETCDF=/home01/pop08/openacc/netcdf
$ export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/home01/pop08/openacc/netcdf/lib:/home01/pop08/openacc/hdf5/lib:/home01/pop08/openacc/zlib/lib

./configure
3
$ vi configure.wrf
첨부한 configure.wrf로 바꿈
line130,132,515의 cuda5.5를 cuda6.0으로 바꿈
line210,215의 -L/shared/apps/rhel-6.2/libs/numactl-2.0.9/lib64 을 삭제
line221,222의 경로를 /home01/pop08/openacc/netcdf로 바꿈
mpicc를 /applic/internal/testbed/pgi/linux86-64/2014/mpi/openmpi-1.8.1/bin/mpicc로 변경
DM_FC,DM_CC에 달린 옵션 삭제
line178의 CFLAGS의 제일 뒤에 -DMPI2_SUPPORT 추가

RUN WRF
$ vi ~/.bashrc 의 마지막에 다음을 추가
exporthost=`/bin/hostname`
if [$host == 'login-tesla' -o ${host:0:5} == 'tesla']
then
export PATH=/usr/bin:/applic/internal/testbed/pgi/linux86-64/2014/mpi/openmpi-1.8.1/bin:$PATH
export LD_LIBRARY_PATH=/home01/pop08/openacc/zlib/lib:/home01/pop08/openacc/hdf5/lib:/home01/pop08/openacc/netcdf/lib:/applic/internal/testbed/pgi/linux86-64/2014/mpi/openmpi-1.8.1/lib:$LD_LIBRARY_PATH
fi

$source~/.bashrc

$ vi hostfile
tesla01slots=20

```
tesla02slots=20
tesla03slots=20
......
tesla12slots=20
$ mpirun -np 12 -hostfile ./hostfile --map-by ppr:1:node ./real.exe
$mpirun-np12-hostfile./hostfile--map-byppr:1:node./wrf.exe
```

Tesla-host

READY

```
$ module purge
$source/applic/internal/testbed/pgi/linux86-64/14.7/pgi.sh
$source/applic/internal/testbed/pgi/linux86-64/14.7/mpi.sh
```

COMPILE WRF

```
$ export NETCDF=/home01/pop08/openacc/netcdf
$ export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/home01/pop08/openacc/netcdf/lib:/home01/pop08/openacc/hdf5/lib:/home01/pop08/openacc/zlib/lib

./configure
3
$ vi configure.wrf
```

mpicc를 /applic/internal/testbed/pgi/linux86-64/2014/mpi/openmpi-1.8.1/bin/mpicc로 변경

DM_FC,DM_CC에 달린 옵션 삭제

line178의 CFLAGS의 제일 뒤에 -DMPI2_SUPPORT 추가

RUN WRF

$ vi ~/.bashrc 의 마지막에 다음을 추가

```
exporthost=`/bin/hostname`
if [ $host == 'login-tesla' -o ${host:0:5} == 'tesla' ]
then
export PATH=/usr/bin:/applic/internal/testbed/pgi/linux86-64/2014/mpi/openmpi-1.8.1/bin:$PATH
export LD_LIBRARY_PATH=/home01/pop08/openacc/zlib/lib:/home01/pop08/openacc/hdf5/lib:/home01/pop08/openacc/netcdf/lib:/applic/internal/testbed/pgi/linux86-64/2014/mpi/openmpi-1.8.1/lib:$LD_LIBRARY_PATH
fi
```

```
$source~/.bashrc

$ vi hostfile
tesla01slots=20
tesla02slots=20
tesla03slots=20
......
tesla12slots=20
$ mpirun -np 12 -hostfile ./hostfile --map-by ppr:1:node ./real.exe
$mpirun-np12-hostfile./hostfile--map-byppr:1:node./wrf.exe
```