인턴십 프로그램 결과 보고서

제　목 : 병렬처리 Platform 구축

기　관 : (주) 쎄트렉아이

부　서 : 지상 3팀

기　간 : 2017.06.19 ~ 2017.08.11 (8주)

졸업연구 대체 신청 : 신청(V)　　미신청( )

학 과 :　전산학부

학 번 :　20130690

성 명 :　한수민　(인)

ㅇ 보고서 요약(실습 일정별로 간략하게 요약, A4용지 3매 이내)

이 프로젝트에서는 Docker라는 가상 이미지 기술을 활용하여 쉽고 빠르게 CPUs와 GPU를 사용하여 계산하는 Hybrid 병렬처리 플랫폼을 구축하였다. 이후 만들어진 Docker 이미지를 실제 Host에 구동하여 개략적인 플랫폼을 구축하고 Slurm이라는 Scheduler를 활용하여 효율적으로 각 노드들을 관리하고 자원을 분배 할 수 있는 환경을 마련하였다. 또한 Program Profiling 도구들을 활용하여 Job의 크기를 정량적으로 계산함으로써 Scheduler를 효율적으로 개선하는 알고리즘에 대해서도 고찰해 보았으며, 실시간 Monitoring tool인 Ganglia, 그리고 특히 Cgroup을 조사함으로써 프로세스에 대해 자원 사용량에 제한을 두고 시스템 자원을 비교적 간단히 관리할 수 있음을 알 수 있었다. 이번 프로젝트를 통해 다소 생소했던 HPC 분야에 대해 공부해보고 병렬처리 플랫폼을 프로토타이핑하는 경험을 할 수 있었으며, kernel과 H/W 적으로 직접적으로 맞닿아 있는 부분에 대해서도 간접적으로 공부함으로써 이후 연구해 나갈 분야나 주제에 대해서도 생각해 볼 수 있는 기회가 되었다.

인턴은 아래와 같은 일정으로 진행되었다.

주차	내용
1	CPUs + GPU Parallel Computing Platform using Docker
2	Search Profiling and Monitoring Tools
3	Profiling Tool Installation & Make into a Dockerfile OpenCL C++ Wrapper / C++ to C Interpolation
4	Slurm Installation
5	Dockerfile with Profiling Tools Installed
6	Slurm on the Real Host Network
7	Cgroup
8	Report & Final Presentation

전체적으로 1~4주 동안 했던 내용은 Docker의 개념을 익히고 가상 이미지를 만들어 여러가지 소프트웨어를 설치해 보고 Trouble Shooting을 하는 단계였다. 하나로 만들어진 Docker 이미지를 여러대의 노드에 배포할 예정이었기 때문에 이미지가 간소화 될 필요가 있었다. 따라서 가급적 이미지의 용량이 작으면서도 필요한 기능은 모두 구동될 수 있는 CentOS 기반의 Docker 이미지를 제작하였으며 Docker는 최근에 국내에서도 각광을 받으며 인기가 많아서 개발과정에서 참고 할 수 있는 국내/외 자료가 많았다. 그래서 비교적 계획에 차질없이 업무를 진행할 수 있었다.

반면 5~8주 동안 했던 내용은 전반기에 제작하였던 Docker 이미지를 실제 노드들에 연결하여 물리적으로 HPC환경을 구축해 보는 것이었다. 그리고 이 과정에서 Cgroup이라는 linux kernel 프로그램에 대해서도 공부해야할 필요가 있었다. 그러나 Slurm이나 Cgroup과 관련된 자료가 매우 부족했고 한국어 자료는 물론 영어로 된 예제나 정리된 문서마저 부족했기에 제대로 된 설치법이나 사용법에 대해서도 익숙해 지기가 어려웠다. 그러나 그렇기에 국내의 전문가가 드물어 블루오션이라는 생각이 들기도 하였다. 분야가 어려운 만큼 충분히 경험을 쌓고 노하우를 가진다면 앞으로 발전해 나가는 HPC분야에서 경쟁력을 가질 수 있을 것이라고 생각들었다.

이 보고서는 아래의 목차로 이루어져있다.

3. Scheduler: Slurm
4. Cgroup
5. Program Profiling Tools
6. Conclusion

먼저 리눅스 가상 이미지 기술인 Docker의 개요에 대해 알아본 뒤, 가상이미지를 활용한 병렬화 시스템 플랫폼을 디자인한다. 이후 Job scheduler인 Slurm에 대해 알아보고, 리눅스에서 프로세스 단위로 자원을 할당하고 관리할 수 있는 Cgroup(control group)에 대해 간략하게 소개한다. 이후 process를 프로파일링 하기위한 Tool들을 설치하고 직접 실험 해 보고, 최종적으로 보고서의 결론과 힘들었던 점, 앞으로의 연구 진행방향 등에 대해서 토의 해 볼 것이다.

또한 보고서 가장 뒷 페이지에 최종 발표 때 사용하였던 발표자료를 함께 첨부하였다. 일종의 요약본과 같은 형태로 비교적 간추려진 형태로 내용을 살펴볼 수 있다.

ㅇ 본 문(A4용지 10매 이상)

[다음 페이지]

병렬 처리 플랫폼 개발

Parallel Processing Platform Development

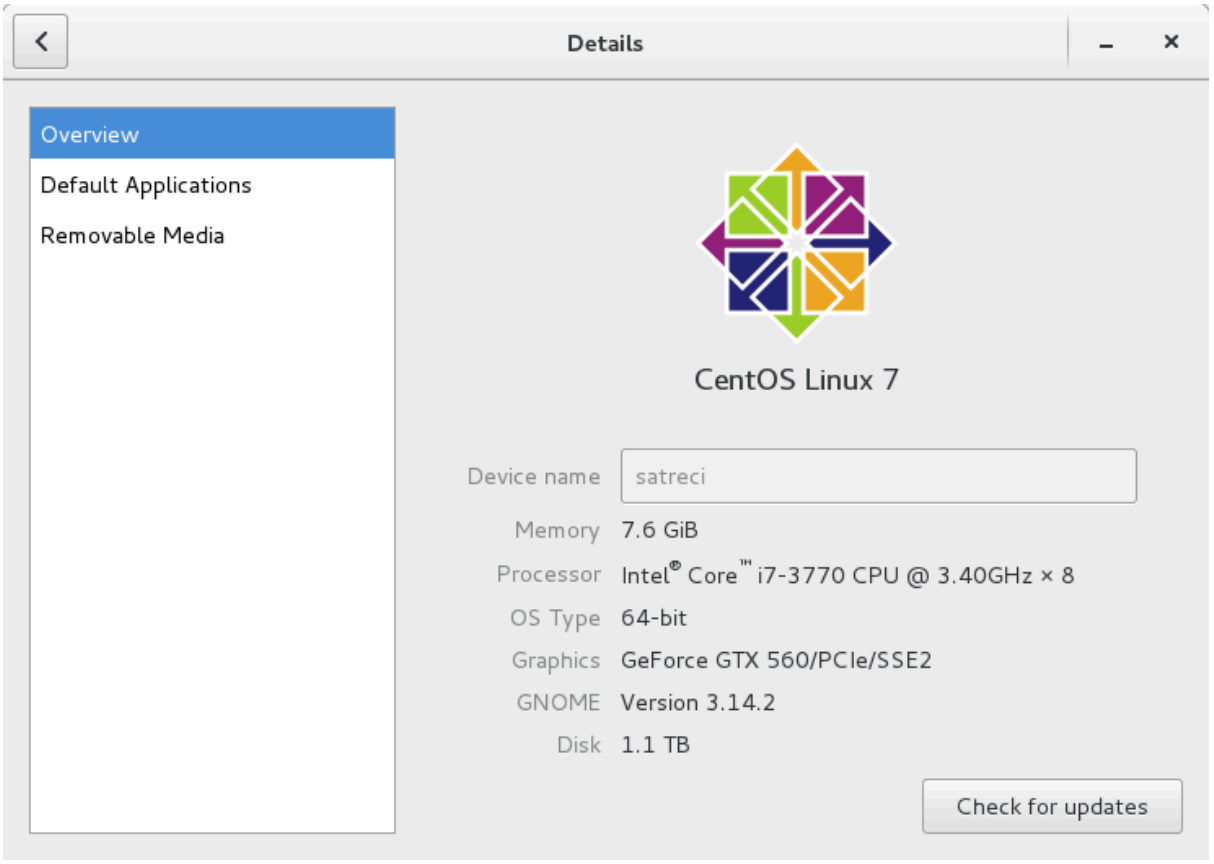
한수민 Sumin Han (hsm6911@kaist.ac.kr)
(주) 쎄트렉아이 CUop 인턴 (2017. 6. 19. ~ 2017. 8. 11. [8주])

Introduction

프로그램의 연산 속도를 높이기 위해 단순하게는 더 좋은 하드웨어를 사용하는 방법이 있지만, 단일 하드웨어는 성능이 높아질 수록 가격이 기하급수적으로 높아지기 때문에 좋은 하드웨어만을 고집하는 것은 비효율적이며 한계를 가진다. 반면 병렬화 기술인 MPI(Massage Passing Interface; 메세지 전달 인터페이스)를 통해 여러대의 CPU를 병렬화 하여 사용하거나 OpenCL(Open Computing Language)로 GPGPU(General-Purpose computing on Graphics Processing Units, GPU 상의 범용 계산)를 사용하여 계산 성능을 높일 수 있다. 최근 빅데이터 시대와 맞물려 머신러닝이나 블록체인(가상화폐) 기술에 사용되기 위해 병렬화에 대한 관심도 높아지고 있는 추세이다. 이번 프로젝트에서는 여러 대의 CPU와 GPU를 연결하여 병렬화 계산을 할 수 있는 Platform의 기반을 개발하였다. 특히 Docker 라는 가상 이미지 기술을 이용하였는데, Docker는 내가 원하는 OS 기반에서 원하는 Software가 설치된 환경을 이미지의 형태로 쉽게 배포할 수 있다. 따라서 OpenMPI나 OpenCL의 설치 환경, 그리고 프로그램을 프로파일링(Profiling)하기 위한 Tool들을 설치한 OS이미지를 배포하여 호환성에 문제 없이 다른 Docker가 설치된 컴퓨터에서도 똑같이 사용할 수 있다. 이 글에서는 병렬 처리 프레임워크를 Docker 이미지를 통해 구축하는 설치 과정과 프로파일링을 위한 Tool들의 설치 및 사용 방법에 대해 정리하였으며 한계점과 개선방향에 대해 고찰해 보았다. 또한 이 글은 CentOS의 운영체제와, NVIDIA GPU를 기반으로 만드는 Docker 이미지에 대해 설명한다.

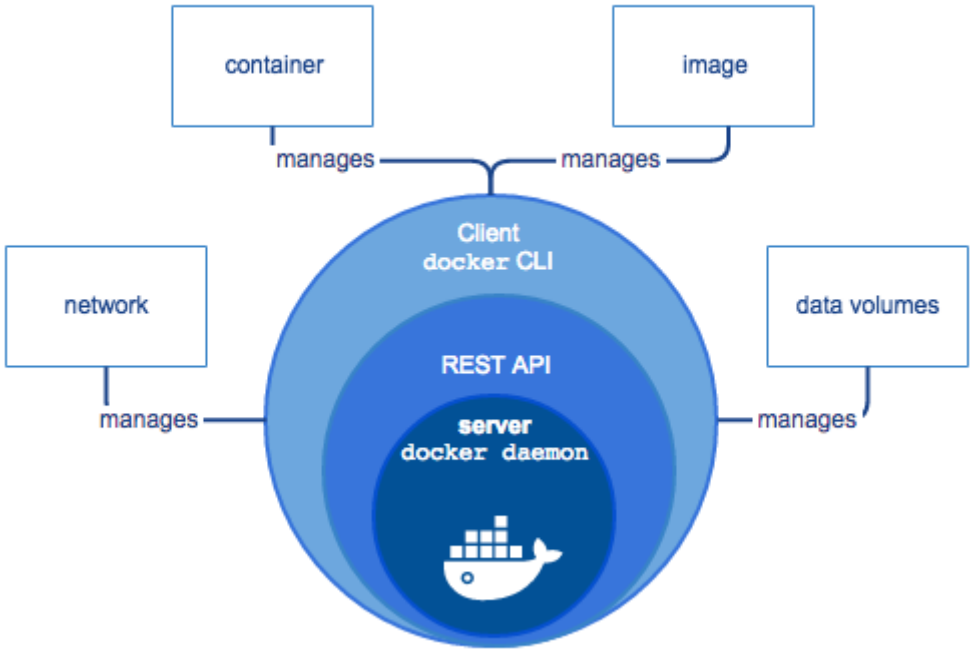
Arthur Keywords: HPC(High Performance Computing); MPI; OpenCL; Multi-core; GPGPU; Docker; System Profiling; Parallel Computing;

Spec:



- **Memory:** 7.6 GiB
- **Processor:** Intel Core i7-3770 CPU @ 3.40GHz x 8
- **OS Type:** 64-bit
- **GPU:** NVIDIA GeForce GTX 560
- **GNOME:** Version 3.14.2
- **Disk:** 1.1 TB

1. What is Docker?



어떤 특수한 환경에서 컴파일 된 프로그램을 다른 컴퓨터에서 똑같이 실행하기 위해서는 OS나 하드웨어의 종류, 설치된 소프트 웨어들의 버전 등을 모두 고려해 주어야 한다. 따라서 같은 프로그램 일지라도 설치된 환경을 모두 고려해 주어야 하는 상황이 생긴다. 이러한 상황을 Docker를 통해 쉽게 해결가능하다. 예를 들어 Linux에서 돌아가는 프로그램 코드를 Windows에서도 Docker라는 가상 이미지 위에 Linux를 돌림으로써 프로그램을 실행할 수 있게 되는 것이다. 물론 가상 이미지를 실행하는 방식이기 때문에 직접적으로 하드웨어를 건드리는 것 보다는 delay가 조금 더 생기는 단점이 있다. 하지만 균일하지 않은 여러 대의 컴퓨터를 병렬화하여 사용하고 싶은 경우에는 Docker를 통해 같은 실행환경을 가진 이미지 파일을 배포하고 공유하여, 병렬 처리 환경을 쉽게 구축할 수 있다.
덧붙여, 시스템 프로파일링(System Profiling)도 HPC 분야에서 중요한 부분을 차지한다. 내가 실행하는 프로그램이 얼마나 효율적으로 구동되는지, 혹시나 memory leak나 deadlock과 같이 시간을 잡아먹는 부분은 없는지 등을 정밀하게 조사할 수 있는 tool들이 필요하기 때문이다. 그러나 이러한 Profiling Tool들을 설치함에 있어서도 OS에 영향을 받고 새로운 환경에 맞게 새롭게 컴파일 하여 설치해 주어야 하는 문제가 발생한다. 이러한 경우에도 내가 원하는 Profiling Tool이 정상적으로 작동하는 이미지가 있으면 이를 쉽게 배포하여 얼마든지 사용할 수 있다. 따라서 사용자는 설치 시 발생하는 문제들에 대해 Trouble Shooting 할 필요 없이 시스템 환경을 구축할 수 있다.
끝으로 쉽게 Docker Image를 배포하기 위해서는 용량을 최소한으로 만드는 작업이 중요하다. 따라서 이들을 효율적으로 설치하고 환경변수를 설정해 주는, 다시말해 Docker image를 생성하는 dockerfile을 사용자가 참고하여 새로운 이미지를 만들 수 있도록 배포하는 작업도 필요하다. 이를 이용해 사용자는 자신의 컴퓨터 환경에 좀 더 customized 된 docker image를 생성할 수 있다.

1.1. Docker Installation

Docker 프로그램은 아래의 command로 설치 가능하다.

```
$ yum -y install docker
```

이후 실질적으로 Docker를 사용하기 위해서는 docker daemon을 아래와 같이 실행해 주어야 한다.

```
$ service docker start
Redirecting to /bin/systemctl start  docker.service
```

1.2. Docker Commands

Docker 에서 자주 사용되는 주요 Command 들에 대해 소개한다.

1.2.1. Pull Docker Image

```
$ docker pull centos:latest
```

최초로 시작할 OS를 받아올 때 docker pull로 온라인 repository 에서 받아올 수 있다. centos, debian, ubuntu등 official한 OS 뿐만 아니라 개인이 만든 docker image도 주소를 입력하여 다운로드 받을 수 있는 명령어이다.

1.2.2. Check Docker Image

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
slurm-tau           latest              79e9af1f33ce        2 days ago          2.011 GB
slurm-light         latest              a4b85dec2c78        3 days ago          984.8 MB
docker.io/centos    7                   36540f359ca3        2 weeks ago         192.5 MB
docker.io/debian    latest              a2ff708b7413        4 weeks ago         100.1 MB
```

docker images 커맨드로 현재 어떠한 이미지가 제작되어 있는지, 용량은 어떻게 되는지 등을 확인 할 수 있다.

1.2.3. Remove Docker Image

docker image는 rmi 명령어로 삭제 가능하다.

```
$ docker rmi <image_name:tag>
```

1.2.4. Docker Run

```
$ docker run --rm -it -h mhost --name cname -v /kernaldata:/dockerdata --env MYVAR=foo --add-host nn:172.17.0.2 -p 2222:22 <image_name> <command(optional)>
```

docker run 으로 실질적으로 이미지를 컨테이너에 불러 실행할 수 있다. 이 때 몇 가지 option을 줄 수 있다.

- --rm 옵션은 종료시 자동으로 컨테이너를 종료한다.

- -i 옵션은 interactive mode를 의미하며 실질적으로 input을 할 수 있다.
- -t 옵션은 tty mode를 의미하며 결과를 터미널에 볼 수 있다. 따라서 -i명령어와 합쳐져 -it 옵션으로 container에서 커맨드를 입력할 수 있게 된다.
- -d 옵션은 데몬모드로 background에서 실행되도록 한다. 다만 특별히 돌아가는 프로그램이 없을 경우 자동으로 exit 된다.
- -h=<host_name> 옵션으로 host 이름을 설정하여 실행 가능하다.
- --name=<container_name> 옵션으로 컨테이너 이름을 설정하여 실행 가능하다.
- -v /kernaldata:/dockerdata 옵션으로 kernal에서의 특정 directory를 docker에서 공유하여 사용할 수 있다.
- --env MYVAR=foo 옵션으로 MYVAR의 환경변수를 foo로 등록하여 실행할 수 있다. 특히 PATH나 LD_LIBRARY_PATH등에 사용된다.
- --add-host <host>:<ip> 옵션으로 host 이름과 ip를 등록하여 /etc/hosts에 설정 되도록 하여 실행할 수 있다.
- -p 2222:22 옵션으로 docker의 port를 publish 할 수 있다 - 이 경우 docker 의 sshd 포트인 22번을 메인 호스트의 2222에 연결하여 외부에서도 2222번 포트로 docker에 sshd 접속을 할 수 있다.

끝으로 image_name을 입력하고 (e.g. centos) 어떤 프로그램을 실행 할 것인지 (e.g. /usr/sbin/sshd, default: /bin/bash)를 설정하여 실행 할 수 있으며, 여러개의 프로그램을 동시에 사용하고 싶을 때는 /bin/bash -c "/usr/bin/supervisord -c /etc/supervisord.conf && gmond && /usr/sbin/sshd -D" 와 같은 방식으로 /bin/bash의 -c 옵션을 주어 여러개의 프로그램을 실행하는 방식으로 할 수 있다.

1.2.5. Check Docker Processes

```
$ docker ps -a
CONTAINER ID     IMAGE          COMMAND                CREATED          STATUS                      PORTS          NAMES
91256e64244c     centos         "/bin/bash"            4 seconds ago    Exited (0) 3 seconds ago                   tiny_hugle
704dce12aead     slurm-tau      "/bin/bash -c '/usr/b" About a minute ago Up About a minute                        c6
3bb8fdec85d7     slurm-light    "/bin/bash -c '/usr/b" About a minute ago Up About a minute                        c5
f010711f9800     slurm-light    "/bin/bash -c '/usr/b" About a minute ago Up About a minute                        c4
0b331a8f630d     slurm-light    "/bin/bash -c '/usr/b" About a minute ago Up About a minute                        c3
dd10a3d4aefd     slurm-light    "/bin/bash -c '/usr/b" About a minute ago Up About a minute                        c2
6f7613129a15     slurm-light    "/bin/bash -c '/usr/b" About a minute ago Up About a minute                        c1
```

docker ps 명령어로 현재 실행되고 있는 container들의 상태를 확인할 수 있다. 어떤 이미지를 실행하고 있는지, 각각의 이름이나 커맨드 status 등은 어떠한지를 보여준다. 특히 -a 옵션을 붙이면 exit된 docker 까지 모두 보여준다.

1.2.6. Remove Docker Process

```
$ docker rm <container_name>
```

docker rm 뒤 container 이름이나 ID를 입력하여 해당 container를 종료시킬 수 있다. 또한 -f 옵션을 통해 강제적으로 종료하는 것도 가능하다. 그리고 너무 많은 container가 실행되어 있을 때

```
$ docker rm $(docker ps -a -q) -f
```

이 명령어를 통해 한 번에 모든 container 를 종료하는 것도 가능하다.

1.2.7. Docker Commit

```
$ docker commit <container_name> <image_save_as...>
```

위와 같이 docker commit 명령어로 현재 진행중인 container의 상태를 이미지화 시켜 저장할 수 있다.

1.2.8. Docker Build

```
$ docker build -t <image_name> .
```

현재 경로 (.) 에 있는 dockerfile에 대해 <image_name>의 이미지를 build하는 명령어이다 (따라서 이전에 자신이 build하고자 하는 directory로 cd명령어로 들어가 있는 것이 필요하다). 보통 설치가 까다로운 Software를 컴파일 순서에 맞게 script에 미리 입력해 두면 그 순으로 커맨드가 실행되어 프로그램을 설치 하고자 할 때 사용되는 기능이며, 이 프로젝트에서도 최종적으로 Profiling tool들 (Slurm, Ganglia, TAU, Valgrind 등)이 설치된 이미지를 자동적으로 build 하도록 돕는다.

1.2.9. Docker Inspect

아래의 명령어로 현재의 docker 정보를 탐색 할 수 있다.

```
$ docker inspect $INSTANCE_ID(or container name)
```

혹은 format을 붙여(--format 또는 -f)

```
$ docker inspect -f "{{ .NetworkSettings.IPAddress }}" <container>
```

로 특정한 정보, 여기서는 해당 container의 ip정보를 얻는 등의 방식으로 사용할 수 있다.

1.3. NVIDIA Graphic Driver Installation

OpenCL을 사용하기 위해서는 **NVIDIA Graphic** 드라이버를 우선적으로 설치하여야 한다. Windows 환경 에서는 설치 파일을 실행시키면 되지만, CentOS 환경에서는 GUI를 사용할 경우에 특별히 기존의 그래픽 드라이버를 제거하고 새로운 NVIDIA 드라이버를 사용하는 방식으로 설치가 이루어지기 때문에 까다로운 부분이 발생한다.

1.3.1. Download Installer

먼저 http://www.nvidia.co.kr/Download/index.aspx 홈페이지에서 NVIDIA GPU 드라이버를 설치하거나 https://developer.nvidia.com/cuda-downloads 에서 CUDA Toolkit을 다운로드 한다. NVIDIA에서 CUDA App을 개발하기 편리하도록 다양한 Tool들을 제공하기에 후자인 CUDA Toolkit을 설치하는 것도 나쁘지 않다. 그러나 기본적으로 현재 프로젝트는 OpenCL을 기준으로 진행되기 때문에 CUDA를 굳이 설치할 필요는 없으며, 용량을 최소화 하기 위해서는 전자를 택하는 것을 추천한다.

설치 명령어는 아래의 커맨드를 입력하면 된다.

```
$ sh cuda_8.0.61_375.26_linux.run
```

그러나 설치가 바로 진행되지 않고 에러 메시지가 발생하는 것을 볼 것이다. 이 때 발생하는 에러 메시지가 가리키는 주소의 log를 분석해 오류의 원인을 살펴가며 설치하면 수월하게 설치 할 수 있다.

1.3.2. Turn off X Server

우선 가장 처음으로 해야 할 일은 기존의 X server를 끄는 일이다. X Server를 끄기 위해서는 GUI 모드가 아니라 터미널만 떠있는 모드로 재부팅 해 주어야 한다. 이를 아래의 커맨드로 부팅 설정을 가능하며 재부팅 이후 적용된다.

```
$ systemctl set-default multi-user.target
```

1.3.3. Disable Nouveau Driver

그리고 Nouveau driver를 비활성화 해 주어야 한다. 이를 위해서 부팅시 Nouveau driver가 실행되지 않도록 해야 하는데,

/etc/default/grub

```
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="ipv6.disable=1 vconsole.font=latarcyrheb-sun16 vconsole.keymap=ko
crashkernel=auto  rhgb quiet rdblacklist=nouveau"
GRUB_DISABLE_RECOVERY="true"
```

이렇게 GRUB_CMDLINE_LINUX 에서 rdblacklist=nouveau 를 끝에 추가 해주어야 한다. 그리고

```
$ grub2-mkconfig -o /boot/grub2/grub.cfg
```

위의 커맨드로 grub2를 갱신한다.

1.3.4. Install kernel-devel

드라이버를 설치하기 위해서 kernel-devel 이 미리 설치가 되어 있어야 한다. 이는 간단하게 yum 으로 설치 가능하다.

```
$ yum install kernel-devel -y
```

여기까지 진행했다면 reboot 한다.

1.3.4. Run Installer

이제 실질적으로 Installer를 실행하여 설치해 주면 된다. kernel-source의 path를 잘 못 잡는 경우가 있으니 입력해 주는 것이 좋다.

```
$ sh cuda_8.0.61_375.26_linux.run --kernel-source-path /usr/src/kernels/3.10.0-514.21.2.el7.x86_64/
```

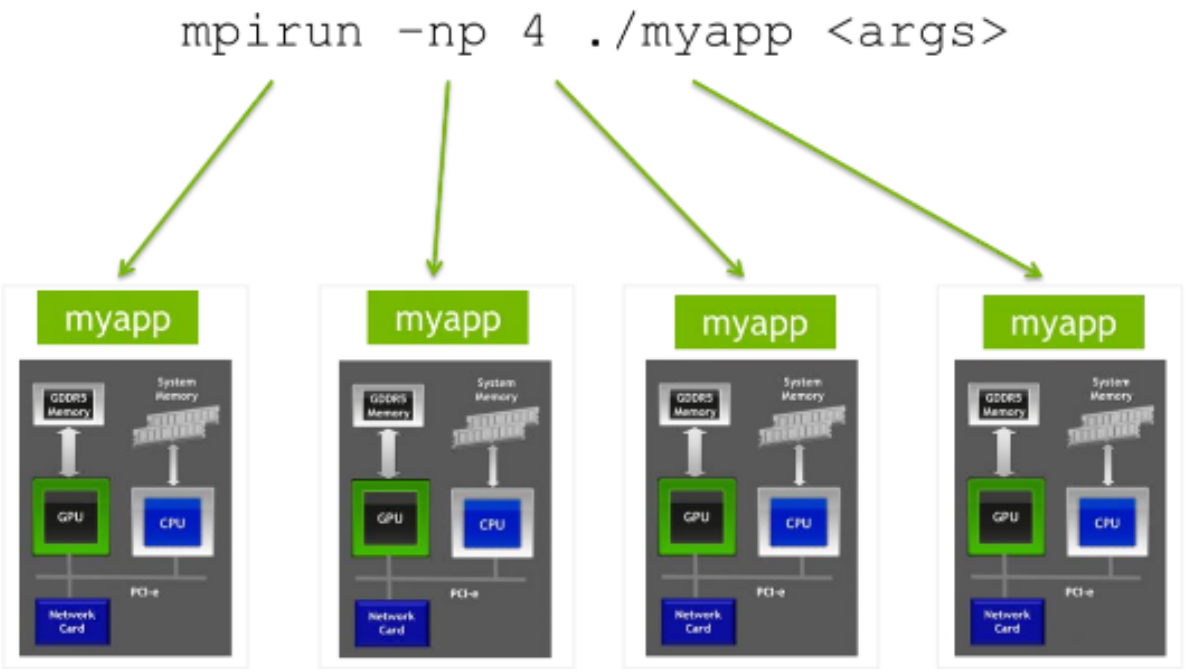
1.3.5. Change back to Graphical Mode

터미널 모드에서 드라이버를 다 설치했다면 다시 graphic 모드로 변경해 주어야 한다.

```
$ systemctl set-default graphical.target
```

1.3.2. 의 과정과 반대로 graphical 모드로 바꾸어 설정하고 reboot 하면 설치가 완료된다.

2. Parallel Computing Platform Design



본격적으로 docker를 활용하여 병렬처리시스템을 구축하고자 한다. 먼저 MPI(메시지 전달 인터페이스)를 활용하여 각 node들에게 연산을 명령하고 그 결과값을 merge하는 방식으로 계산을 병렬화 하여 빠른 시간내에 결과값을 얻을 수 있다. 또한 각각의 node에 설치된 GPU를 OpenCL을 활용하여 multiple GPU를 사용하도록 한다.

2.1. OpenMPI 설치

홈페이지 https://www.open-mpi.org/ 에 들어가 가장 최신버전의 openmpi를 설치한다. 기본적으로 yum으로 설치가 가능하지만 최신 버전을 지원하지 않는 경우가 있기 때문에 manual하게 설치하는 쪽이 더 안전하다.
먼저 아래의 커맨드로 openmpi를 다운로드받아 압축을 풀고 configure하여 install까지 진행한다. 이 때 yum으로 미리 설치 해 주어야 하는 프로그램을 설치한 이후 진행한다.

```
$ yum -y install which tar perl gcc gcc-c++ make openssh-server openssh-clients
$ wget https://www.open-mpi.org/software/ompi/v2.1/downloads/openmpi-2.1.1.tar.gz
$ gunzip -c openmpi-2.1.1.tar.gz | tar xf -
$ cd openmpi-2.1.1
$ ./configure --prefix=/usr/local
<...lots of output...>
$ make all install
```

이후 example 코드[Example: 1]를 컴파일 하고 실행시켜본다.

example1.c

```
 1  #include <mpi.h>
 2  #include <stdio.h>
 3
 4  int main(int argc, char** argv) {
 5      // Initialize the MPI environment
 6      MPI_Init(NULL, NULL);
 7
 8      // Get the number of processes
 9      int world_size;
10      MPI_Comm_size(MPI_COMM_WORLD, &world_size);
11
12      // Get the rank of the process
13      int world_rank;
14      MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
15
16      // Get the name of the processor
17      char processor_name[MPI_MAX_PROCESSOR_NAME];
18      int name_len;
19      MPI_Get_processor_name(processor_name, &name_len);
20
21      // Print off a hello world message
22      printf("Hello world from processor %s, rank %d"
23             " out of %d processors\n",
24             processor_name, world_rank, world_size);
25
26      // Finalize the MPI environment.
27      MPI_Finalize();
28      return 0;
29  }
```

mpicc로 컴파일 하고 mpirun으로 실행한다.

```
$ mpicc example1.c -o example1
$ mpirun -n 4 example1
Hello world from processor satreci, rank 0 out of 4 processors
Hello world from processor satreci, rank 2 out of 4 processors
Hello world from processor satreci, rank 3 out of 4 processors
Hello world from processor satreci, rank 1 out of 4 processors
```

MPI를 설치 했으면 해당 docker를 commit하여 repository로 저장한다. 예를들어 현재 설치한 Docker 의 container 이름이 cname 이었다면

```
$ docker ps
CONTAINER ID    IMAGE     COMMAND       CREATED          STATUS          PORTS     NAMES
99ae1444e4b6    centos    "/bin/bash"   3 seconds ago    Up 2 seconds              cname
$ docker commit cname myimage
```

로 새로운 이미지 이름 myimage 로 저장할 수 있다.

2.2. Password 없는 SSH 설정

MPI 를 여러 docker host를 사용하여 연산하기 위해서는 패스워드가 없는 SSH를 설정 해 주어야 한다. 먼저 2.1.에서 openssh-server 와 openssh-clients를 설치 해 주었는데, 자신의 public key를 server에 등록 해 주고, 이렇게 인증된 docker를 똑같이 배포한다면 모두 같은 key를 갖고있기 때문에 패스워드가 필요없는 SSH 환경을 구축 할 수 있다.

```
$ # OpenSSH Setting (Register Public Key)
$ yum install -y openssh-server openssh-clients
$ ssh-keygen -A
$ ssh-keygen -f $HOME/.ssh/id_rsa -t rsa -N ''
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
$ echo "StrictHostKeyChecking no" >> ~/.ssh/config
```

먼저 ssh-keygen 명령어로 key를 생성할 수 있다. 생성된 public key는 ~/.ssh/id_rsa.pub에 저장되는데, 이것을 ~/.ssh/authorized_keys에 등록하면 해당 key에 대해서 자동 로그인을 할 수 있다. 이후 "StrictHostKeyChecking no"를 ~/.ssh/config에 추가함으로써 패스워드를 묻지 않고 SSH 접속을 등록 된 key 만으로 진행할 수 있다. 이 방법으로 MPI를 사용 할 수 있다.

2.3. Test MPI with multiple hosts

2.1. 에 소개된 example1을 여러개의 docker에서 MPI를 실행하는 방식으로 연산을 해 보자. 먼저 sshd를 실행한 여러개의 docker가 필요하다.
$ docker run -d -v /dockerdata/:/dockerdata/ myimage /usr/sbin/sshd -D
위의 커맨드로 데몬으로 돌아가는 docker container를 실행할 수 있으며, /usr/sbin/sshd로 ssh를 실행 할 수 있다. 특히 -D 옵션을 주어 "/usr/sbin/sshd -D" 를 실행하여 daemon이 자동으로 종료되지 않도록 한다. 또한 -v 옵션에서 /dockerdata/의 경로를 공유하여 mpirun 을 시킬 때 파일을 공유하도록 한다. 먼저 4개 정도의 docker 를 실행시킨 뒤, -it 모드로 mpirun 명령을 수행 할 새로운 docker 하나를 실행하도록 하자.

```
$ docker run -d -v /dockerdata/:/dockerdata/ -h h1 --name c1 myimage /usr/sbin/sshd -D
$ docker run -d -v /dockerdata/:/dockerdata/ -h h2 --name c2 myimage /usr/sbin/sshd -D
$ docker run -d -v /dockerdata/:/dockerdata/ -h h3 --name c3 myimage /usr/sbin/sshd -D
$ docker run -d -v /dockerdata/:/dockerdata/ -h h4 --name c4 myimage /usr/sbin/sshd -D
$ docker run -it /dockerdata/:/dockerdata/ -h hh --name cc myimage /bin/bash
[root@hh]#
```

이제 /dockerdata/ 경로로 들어가 컴파일된 example1을 실행하는데, 이 때 각각의 c1, c2, c3, c4 컨테이너의 ip address를 알아야 한다. 이는 kernel에서 아래의 명령어로 확인 가능하다.

```
$ docker inspect -f "{{ .NetworkSettings.IPAddress }}" c1
172.17.0.2
$ docker inspect -f "{{ .NetworkSettings.IPAddress }}" c2
172.17.0.3
$ docker inspect -f "{{ .NetworkSettings.IPAddress }}" c3
172.17.0.4
$ docker inspect -f "{{ .NetworkSettings.IPAddress }}" c4
172.17.0.5
```

다시 docker 돌아가서 mhostfile 이라는 파일을 하나 생성하여 위의 ip 주소들을 입력한다.

mhostfile

1	172.17.0.2
2	172.17.0.3
3	172.17.0.4
4	172.17.0.5

최종적으로 mpirun을 실행한다.

```
[root@hh]# mpirun --allow-run-as-root -n 4 --hostfile mhostfile example1
Warning: Permanently added '172.17.0.5' (ECDSA) to the list of known hosts.
Warning: Permanently added '172.17.0.3' (ECDSA) to the list of known hosts.
Warning: Permanently added '172.17.0.2' (ECDSA) to the list of known hosts.
Warning: Permanently added '172.17.0.4' (ECDSA) to the list of known hosts.
Hello world from processor h1, rank 0 out of 4 processors
Hello world from processor h3, rank 2 out of 4 processors
Hello world from processor h4, rank 3 out of 4 processors
Hello world from processor h2, rank 1 out of 4 processors
```

다만 최초 실행시에 한하여 위의 결과처럼 Warning이 발생할 수 있으며 이는 ssh 연결 과정에서 발생한 것이기 때문에 문제없다.

2.4. NVIDIA Docker, OpenCL Installation

Docker에서 NVIDIA GPU를 인식하여 사용하기 위해서는 NVIDIA-Docker를 통해 이미지를 실행 해 주어야 한다. 설치 이후 docker 실행 명령어와 사용법이 똑같은 nvidia-docker로 docker 이미지를 실행하여 container에서도 GPU를 사용 할 수 있게 된다. (https://github.com/NVIDIA/nvidia-docker)

```
$ # Install nvidia-docker and nvidia-docker-plugin
$ wget -P /tmp
https://github.com/NVIDIA/nvidia-docker/releases/download/v1.0.1/nvidia-docker-1.0.1-1.x86_64.rpm
$ sudo rpm -i /tmp/nvidia-docker*.rpm && rm /tmp/nvidia-docker*.rpm
$ sudo systemctl start nvidia-docker
```

위의 설치를 완료하였으면 아래의 dockerfile을 build 하여 nvidia-docker 로 실행시켜본다.

Dockerfile

```
 1  FROM centos:7
 2  LABEL maintainer "NVIDIA CORPORATION <cudatools@nvidia.com>"
 3
 4  LABEL com.nvidia.volumes.needed="nvidia_driver"
 5
 6  RUN yum -y update && yum install -y epel-release ocl-icd clinfo
 7
 8  RUN mkdir -p /etc/OpenCL/vendors && \
 9      echo "libnvidia-opencl.so.1" > /etc/OpenCL/vendors/nvidia.icd
10
11  RUN echo "/usr/local/nvidia/lib" >> /etc/ld.so.conf.d/nvidia.conf && \
12      echo "/usr/local/nvidia/lib64" >> /etc/ld.so.conf.d/nvidia.conf
13
14  ENV PATH /usr/local/nvidia/bin:/usr/lib64/openmpi/bin/:${PATH}
15  ENV LD_LIBRARY_PATH /usr/local/nvidia/lib:/usr/local/nvidia/lib64
```

위의 dockerfile은
https://github.com/NVIDIA/nvidia-docker/blob/v1.0.0/ubuntu-14.04/opencl/runtime/Dockerfile 에서 Ubuntu 기반으로 만들어진 dockerfile을 centos에 맞도록 수정한 것이며 이 과정에서 OpenCL Binder를 함께 설치하게 된다. (ocl-icd, clinfo)
이후 dockerfile로 저장한 이후에 해당 디렉토리로 들어가 아래 명령어로 build 해 준다.

```
$ docker build -t ocltest .
```

이후 아래처럼 **nvidia-docker**를 사용하여 해당 이미지를 실행할 때 nvidia-smi를 실행하도록 하여 GPU정보를 제대로 인식하는지를 확인해 보자.

```
$ nvidia-docker run -it --rm ocltest nvidia-smi
Tue Jul 25 06:38:01 2017
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 375.26                 Driver Version: 375.26                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 560     Off  | 0000:01:00.0     N/A |                  N/A |
| 19%   36C   P12    N/A /  N/A |    242MiB /   961MiB |     N/A      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
```

```
|=============================================================================|
|    0                  Not Supported                                         |
+-----------------------------------------------------------------------------+
```

2.5. 최종 Dockerfile 생성

Section 2.1. ~ 2.4. 까지 진행한 내용을 하나의 dockerfile로 만들어 배포하기 쉽게 만들었다. 즉 위의 설치과정을 자동화 하여 동일한 기능의 이미지를 생성해 준다.

Dockerfile

```
# Docker Image
# OpenMPI + OpenCL (NVIDIA) + Slurm + Ganglia
FROM centos7
MAINTAINER "Sumin Han @ Satreci (KAIST internship)"
LABEL com.nvidia.volumes.needed="nvidia_driver"

RUN yum -y install epel-release
RUN yum -y update

# OpenMPI Installation
RUN yum install -y which tar perl gcc gcc-c++ make
RUN curl -O https://www.open-mpi.org/software/ompi/v2.1/downloads/openmpi-2.1.1.tar.gz \
    && gunzip -c openmpi-2.1.1.tar.gz | tar xf - \
    && cd openmpi-2.1.1 \
    && ./configure --prefix=/usr/local \
    && make all install
RUN rm -r -f openmpi-2.1.1.tar.gz openmpi-2.1.1

ENV PATH /usr/local/bin:$PATH
ENV LD_LIBRARY_PATH /usr/local/lib:$LD_LIBRARY_PATH

# OpenCL Setting
RUN yum install -y ocl-icd clinfo

RUN mkdir -p /etc/OpenCL/vendors && \
    echo "libnvidia-opencl.so.1" > /etc/OpenCL/vendors/nvidia.icd

RUN echo "/usr/local/nvidia/lib" >> /etc/ld.so.conf.d/nvidia.conf && \
    echo "/usr/local/nvidia/lib64" >> /etc/ld.so.conf.d/nvidia.conf

ENV PATH /usr/local/nvidia/bin:${PATH}
ENV LD_LIBRARY_PATH /usr/local/nvidia/lib:/usr/local/nvidia/lib64:$LD_LIBRARY_PATH

# OpenSSH Setting (Public Key registeration)
RUN yum install -y openssh-server openssh-clients
RUN ssh-keygen -A
RUN ssh-keygen -f $HOME/.ssh/id_rsa -t rsa -N ''
RUN cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
RUN echo "StrictHostKeyChecking no" >> ~/.ssh/config
```

2.6. Run Sample Code

실질적으로 MPI와 OpenCL을 사용하는 프로그램을 만들어 테스트 해 보자. 먼저 OpenCL 프로그램을 컴파일 하기 위해서는 opencl-headers와 ocl-icd-devel이 설치되어 있어야 한다.
$ yum -y install opencl-headers ocl-icd-devel
이후 gcc나 mpicc 등을 이용해 컴파일 할 때 -lOpenCL 을 뒤에 붙여주면 된다.
여기서 소개할 예제는 Simpson 적분 공식을 OpenCL과 MPI를 이용하여 계산하는 코드이며 한국 슈퍼컴퓨팅 경진대회 2015년 1번 문제이다.
Reference: https://en.wikipedia.org/wiki/Simpson%27s_rule
Smpson 적분공식을 활용한 1차원 정적분 프로그램을 고려한다. 주어진 시작 프로그램(C 또는 FORTRAN)을 병렬화(MPI/OpenMP)시켜서 주어진 시스템에서 적분값과 총 함수 호출 횟수를 출력하시오. 또한, 최대 병렬 효율 성능을 구하시오. 여기서 사용된 Simpson 정적분 공식은 구간 [a,b]에 대한 정적분을 의미하고, n+1개의 함수값을 호출하게 되어 있다. 여기서 n은 짝수 이다. 실제 정적분값은 다음의 수식으로 근사된다. (h/3)[f(x0)+4f(x1)+2f(x2)+....+4f(xn-1)+f(xn)]. 여기서, h=(b-a)/n 이며, xi=a+ih이고, f(x)=4/(1+x2)이다. 주의할 점은 함수 호출을 n+1번만 수행한다.
Simpson 적분공식은

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \ldots + 4f(x_{n-1}) + f(x_n)\right]$$

이와 같다. 다시말해, x0와 xn을 제외한 부분을 1차 FDM으로 계산하면 된다.
대회측에서 제시한 MPI만을 사용한 병렬화 답안 코드는 아래와 같다.

simpson-mpi.c

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<math.h>
4
5  #include<mpi.h>
6
7  #define min(a,b) ((a)<(b)?(a):(b))
8
9  int ncall;
10
11 double simpson2(double (*func1)(double),int , double , double , int , int );
12
13 double func(double x){
14     ncall ++;
15     return 4.L/(x*x+1.L);
16 }
17
18 int main(int argc, char **argv){
19     double rslt,aa,bb,rslt0;
20     int n,ncall0;
21     int ierr, kount, iroot;
22     int myid,nproc;
23     double time1,time2,time_start, time_end;
24
25     MPI_Init(&argc, &argv);
26     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
27     MPI_Comm_size(MPI_COMM_WORLD,&nproc);
```

```
28
29      if(myid ==0 && nproc > 1) printf("%d processes are alive\n",nproc);
30      if(myid ==0 && nproc == 1) printf("%d processes is alive\n",nproc);
31
32      time_start = MPI_Wtime();
33      n = 20;
34      n = 200000;
35      n = 2000000000;
36
37      aa = 0.L;
38      bb = 1.L;
39      ncall = 0;
40      ncall = 0;
41
42      rslt = simpson2(func,n,aa,bb,myid,nproc);
43
44      iroot = 0; kount = 1;
45      MPI_Reduce(&rslt,&rslt0, kount, MPI_DOUBLE, MPI_SUM,iroot, MPI_COMM_WORLD);
46      if(myid==0) printf("%d %18.14g n, rslt0\n",n,rslt0);
47      iroot = 0; kount = 1;
48      MPI_Reduce(&ncall, &ncall0,kount,MPI_INT,MPI_SUM,iroot, MPI_COMM_WORLD);
49      if(myid==0) printf("%d ncall0\n",ncall0);
50
51      time_end = MPI_Wtime();
52      if(myid==0) {
53          double timer = time_end - time_start;
54          double timerm = timer/60.L;
55          double timerh = timer/3600.L;
56          double timerd = timer/3600.L/24.L;
57          printf("%14.5g s %14.5g m %14.5g h %14.5g d\n",timer, timerm,timerh,timerd);
58      }
59      MPI_Finalize();
60      return 0;
61  }
62
63  void equal_load(int n1, int n2, int nproc, int myid, int *istart, int *ifinish){
64      int iw1,iw2;
65      iw1 = (n2-n1+1)/nproc;
66      iw2 = (n2-n1+1)%nproc;
67      *istart = myid*iw1+n1+min(myid,iw2);
68      *ifinish= *istart +iw1 -1;
69      if(iw2>myid) *ifinish = *ifinish + 1;
70      if(n2<*istart) *ifinish = *istart - 1;
71  }
72
73  double simpson2(double (*func1)(double),int n, double aa, double bb, int myid, int nproc){
74      double rslt;
75      double h,xx;
76      int j,n1,n2;
77      int istart,ifinish;
78
79      rslt = 0.;
80      if((n%2) != 0) {
81          fprintf(stderr,"input error, n must be even number %d\n",n);
82          exit(9);
```

```
 83      }
 84
 85      n1 = 1; n2 = n-1;
 86      equal_load(n1,n2,nproc,myid,&istart,&ifinish);
 87
 88
 89      h = (bb-aa)/(double)n;
 90      if(myid==0) rslt = (func1(aa)+func1(bb));
 91      for(j=istart;j<=ifinish;j++){
 92          xx = aa + h*(double)j;
 93          if((j&1)) {
 94              rslt += 4.L*func1(xx);
 95          }
 96          else {
 97              rslt += 2.L*func1(xx);
 98          }
 99      }
100      rslt = rslt * h/3.L;
101      return rslt;
102  }
```

위의 코드는 n의 값을 조절 해 가며 테스트 해 볼 수 있으며, 결과된 계산값은 n이 클 수록 pi=3.14159265359 값으로 수렴한다. n = 2,000,000,000 기준 현재 컴퓨터 스펙으로 40 초 가량 계산시간이 걸린다.

```
[root@rndc smpi]# ./a.out
1 processes is alive
2000000000    3.1415926535898 n, rslt0
2000000001 ncall0
      41.809 s       0.69682 m       0.011614 h      0.0004839 d
[root@rndc smpi]# mpirun --allow-run-as-root -n 8 a.out
8 processes are alive
2000000000    3.1415926535897 n, rslt0
2000000001 ncall0
      8.6967 s       0.14494 m      0.0024157 h     0.00010066 d
```

(위의 계산은 kernel에서 실행 된 것이며 여러 대의 docker를 활용한 계산은 2.3. 을 참고하면 된다.)
지금의 MPI만을 사용하는 코드를 개선하여 각 Docker가 GPU를 사용하여 계산된 결과 값을 merge하는 방식, 즉 CPU와 GPU의 Hybrid한 병렬화 시스템으로 계산하기 위한 예제를 디자인 하였다. OpenCL을 사용하여 Simpson's Rule을 적용한 계산은 아래와 같다.

simpson-mpicl.cl

```
1   __kernel
2   void func(__global float *X,
3             __global float *Y)
4   {
5       // Get the work-item's unique ID
6       int idx=get_global_id(0);
7       float dn = X[idx]*X[idx] + 1.L;
8       Y[idx] = 4.L  / dn;
9   }
```

simpson-mpicl.c

```
1   // This program tests Simpson theorem using OpenCL
2
3   // MPI Includes
4   #include <mpi.h>
5
6   // System includes
7   #include <stdio.h>
8   #include <stdlib.h>
9
10  // OpenCL includes
11  #include <CL/cl.h>
12
13  #define MAX_SOURCE_SIZE (0x100000)
14  // Define min macro
15  #define min(X, Y) ((X) < (Y) ? (X) : (Y))
16
17  void equal_load(int, int, int, int, int*, int*);
18  int main(int argc, char** argv)
19  {
20      int myid, nproc;
21      int n = 500000, i;
22      float sum = 0.L;
23      float aa = 0.L;
24      float bb = 1.L;
25      float time_start, time_end;
26      float h = (bb-aa)/(float)n;
27      int istart, ifinish;
28
29      // Initialize the MPI environment
30      MPI_Init(NULL, NULL);
31
32      // Get the number of processes
33      MPI_Comm_size(MPI_COMM_WORLD, &nproc);
34
35      // Get the rank of the process
36      MPI_Comm_rank(MPI_COMM_WORLD, &myid);
37
38      // Check starting time
39      time_start = MPI_Wtime();
40
41      // Load the source code containing the kernel
42      FILE *fp;
43      char fileName[] = "./simpson-mpicl.cl";
44      char *source_str;
45      size_t source_size;
46
47      /* Load the source code containing the kernel*/
48      fp = fopen(fileName, "r");
49      if (!fp) {
50          fprintf(stderr, "Failed to load kernel.\n");
51          exit(1);
```

```
52      }
53      source_str = (char*)malloc(MAX_SOURCE_SIZE);
54      source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
55      fclose(fp);
56
57      // Load equally
58      equal_load(1, n-1, nproc, myid, &istart, &ifinish);
59      printf("(%d / %d) calculate from %d to %d\n", myid+1, nproc, istart, ifinish);
60
61      // num elements
62      int nelem = ifinish - istart + 1;
63
64      // This code executes on the OpenCL host
65      // Host data
66      float *X=NULL; // Input array
67      float *Y=NULL; // Output array
68
69      // Compute the size of the data
70      size_t datasize=sizeof(float)*nelem;
71
72      // Allocate space for input/output data
73      X=(float*)malloc(datasize);
74      Y=(float*)malloc(datasize);
75
76      // Initialize the input data
77      for(i=0; i<nelem; i++){
78          X[i]= aa + h*(float)(i+istart);
79      }
80
81      // Use this to check the output of each API call
82      cl_int status;
83
84      printf("A%d\n", myid);
85      // Retrieve the number of platforms
86      cl_uint numPlatforms=0;
87      status=clGetPlatformIDs(0, NULL, &numPlatforms);
88
89      printf("B numPlatforms = %d %d\n", myid, numPlatforms);
90      // Allocate enough space for each platform
91      cl_platform_id *platforms=NULL;
92      platforms=(cl_platform_id*)malloc(
93          numPlatforms*sizeof(cl_platform_id));
94
95      printf("C%d\n", myid);
96      // Fill in the platforms
97      status=clGetPlatformIDs(numPlatforms, platforms, NULL);
98
99      printf("D%d\n", myid);
100     // Retrieve the number of devices
101     cl_uint numDevices=0;
102     status=clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, 0,
103         NULL, &numDevices);
104
105     printf("E%d\n", myid);
106     // Allocate enough space for each device
```

```
107     cl_device_id *devices;
108     devices=(cl_device_id*)malloc(
109         numDevices*sizeof(cl_device_id));
110
111     printf("F%d\n", myid);
112     // Fill in the devices
113     status=clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL,
114         numDevices, devices, NULL);
115
116     printf("G%d\n", myid);
117     // Create a context and associate it with the devices
118     cl_context context;
119     context=clCreateContext(NULL, numDevices, devices, NULL,
120         NULL, &status);
121
122     printf("H%d\n", myid);
123     // Create a command queue and associate it with the device
124     cl_command_queue cmdQueue;
125     cmdQueue=clCreateCommandQueue(context, devices[0], CL_QUEUE_PROFILING_ENABLE,
126         &status);
127
128     // Create a buffer object that will contain the data
129     // from the host array A
130     cl_mem bufX;
131     bufX=clCreateBuffer(context, CL_MEM_READ_ONLY, datasize,
132         NULL, &status);
133
134     // Create a buffer object that will contain the output data
135     // from the host array Y
136     cl_mem bufY;
137     bufY=clCreateBuffer(context, CL_MEM_WRITE_ONLY, datasize,
138         NULL, &status);
139
140     // Write input array A to the device buffer bufferA
141     status=clEnqueueWriteBuffer(cmdQueue, bufX, CL_FALSE,
142         0, datasize, X, 0, NULL, NULL);
143
144     // Write input array A to the device buffer bufferB
145     status=clEnqueueWriteBuffer(cmdQueue, bufY, CL_FALSE,
146         0, datasize, Y, 0, NULL, NULL);
147
148     // Create a program with source code
149     cl_program program=clCreateProgramWithSource(context, 1,
150         (const char**)&source_str, NULL, &status);
151
152     // Build (compile) the program for the device
153     status=clBuildProgram(program, numDevices, devices,
154         NULL, NULL, NULL);
155
156     // Create the vector addition kernel
157     cl_kernel kernel;
158     kernel=clCreateKernel(program, "func", &status);
159
160     // Associate the input and output buffers with the kernel
161     status=clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufX);
```

```
162     status=clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufY);
163
164     // Define an index space (global work size) of work
165     // items for execution. A workgroup size (local work size)
166     // is not required, but can be used.
167     size_t globalWorkSize[1];
168
169     // There are 'nelem' work-items
170     globalWorkSize[0]=nelem;
171
172     cl_event event;
173     // Execute the kernel for execution
174     status=clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL,
175        globalWorkSize, NULL, 0, NULL, &event);
176
177     // Read the device output buffer to the host output array
178     clEnqueueReadBuffer(cmdQueue, bufY, CL_TRUE, 0,
179        datasize, Y, 0, NULL, NULL);
180     // Reduce the output
181     for(i=0; i<nelem; i++) {
182        if ((myid == 0 && i == 0) || (myid == nproc-1 && i == nelem-1)) sum += Y[i];
183        else if ((istart + i)%2 == 0) sum += Y[i] * 2;
184        else sum += Y[i] * 4;
185     }
186
187     sum = sum * h / 3.L;
188
189     if(nproc > 1) {
190        if(myid > 0) { // Slave
191           MPI_Send(&sum, 1, MPI_FLOAT, 0, 42, MPI_COMM_WORLD);
192        }
193        else { // Master
194           float total = sum;
195           float tmp;
196           for(i=1; i<nproc; i++){
197              MPI_Recv(&tmp, 1, MPI_FLOAT, i, 42, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
198              total += tmp;
199           }
200           printf("Result = %f\n", total);
201        }
202     }
203     else {
204        printf("Result = %f\n", sum);
205     }
206
207     time_end = MPI_Wtime();
208     if(myid==0) {
209        float timer = time_end - time_start;
210        printf("%14.5g s\n", timer);
211     }
212
213     cl_ulong t_start, t_end;
214     t_start = t_end = 0;
215     double t_time;
216
```

```
217     printf("%llu, %llu\n", t_start, t_end);
218     clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &t_start, NULL);
219     clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &t_end, NULL);
220     t_time = t_end - t_start;
221     printf("%llu, %llu\n", t_start, t_end);
222     printf("Execution time in milliseconds = %0.3f ms\n", (t_time / 1000000.0) );
223
224     // Free OpenCL resources
225     clReleaseKernel(kernel);
226     clReleaseProgram(program);
227     clReleaseCommandQueue(cmdQueue);
228     clReleaseMemObject(bufX);
229     clReleaseMemObject(bufY);
230     clReleaseContext(context);
231
232     // Free host resources
233     free(source_str);
234     free(platforms);
235     free(devices);
236     free(X);
237     free(Y);
238
239     // Finalize the MPI environment.
240     MPI_Finalize();
241
242     return 0;
243 }
244
245 void equal_load(int n1, int n2, int nproc, int myid, int *istart, int *ifinish) {
246     int iw1, iw2;
247     iw1 = (n2-n1+1)/nproc;
248     iw2 = (n2-n1+1)%nproc;
249     *istart = myid*iw1 + n1 + min(myid, iw2);
250     *ifinish = *istart + iw1 - 1;
251     if (iw2 > myid) *ifinish = *ifinish + 1;
252     if (n2 < *istart) *ifinish = *istart - 1;
253 }
```

요약하자면 .cl 코드에서는 해당 값에 대한 함수를 실행하며, c 코드(main) 에서는 GPU를 실행하기 위한 메모리 작업이나 작업 큐 실행 등에 대한 코드를 갖고 있으며 최종적으로 결과를 merge하는 것을 수행한다. 다만 코드의 효율성은 크게 고려하지 않았고 MPI와 OpenCL을 동시에 사용하기 위한 코드이므로 실행시간이 많이 걸려서 프로파일링 테스트에는 적합하지 않은 예제임을 감안한다.
컴파일 방법은 아래와 같다.

```
$ mpicc simpson-mpicl.c -o prog -lOpenCL
```

결과적으로 multiple docker에 실행하였을 때 아래와 같이 출력된다.

```
root@nn:/dockerdata/examples/clprofsimpson# mpirun --allow-run-as-root -n 4 --hostfile mhostfile prog
(1 / 4) calculate from 1 to 125000
(2 / 4) calculate from 125001 to 250000
(3 / 4) calculate from 250001 to 375000
(4 / 4) calculate from 375001 to 499999
```

```
[Rank 2] Execution time in milliseconds = 0.024 ms
[Rank 4] Execution time in milliseconds = 0.045 ms
[Rank 3] Execution time in milliseconds = 0.023 ms
Result = 3.141596
      0.21094 s
[Rank 1] Execution time in milliseconds = 0.026 ms
```

이 때, Execution time in milliseconds 부분은 OpenCL에서 프로파일링 된 GPU 실행 시간을 의미한다. 여기서 차후에 소개할 GPU Profiling을 위해 기본적으로 OpenCL에서 제공하는 Profiling 기능을 활용하였다 (Line 218~219). 그 이유는 CPU에서는 GPU에 명령을 내린 이후 GPU가 연산하는 동안 waiting 상태에 들어가기 때문에 Profiling을 하지 않으면 GPU에서 각각의 상태(e.g. write data, update kernel, read data, etc)가 어떻게 진행되는지를 모르기 때문이다. 자세한 Profiling에 대해서는 이후 소개한다.

3. Scheduler: Slurm

3.1. Introduction to Slurm

Slurm은 오픈 소스이며 내결함성(fault-tolerant)이고 높은 확장성을 가지는 cluster management 와 job scheduling system이다. 다음의 3가지의 Key function을 가진다.

1. 리소스(컴퓨팅 노드)에 대한 배타적 혹은 비 배타적 접근을 일정 기간동안 사용자에게 할당하여 작업을 수행 할 수 있다.
2. 할당된 노드 집합에서 작업(일반적으로 병렬 작업)을 시작, 실행 및 모니터링하기 위한 프레임워크를 제공한다.
3. 보류중인 작업 대기열을 관리하여 자원에 대한 경합을 중재한다.

3.2. Slurm Installation

설치를 위해서는 먼저 아래의 dockerfile, slurm.conf, supervisord.conf, gmond.conf 를 같은 경로에 저장한 이후 build한다.

3.2.1. Create Dockerfile

Dockerfile

```
 1  # Docker Image
 2  # OpenMPI + OpenCL (NVIDIA) + Slurm + Ganglia
 3  FROM centos
 4  MAINTAINER "Sumin Han @ Satreci (KAIST internship)"
 5  LABEL com.nvidia.volumes.needed="nvidia_driver"
 6
 7  RUN yum -y install epel-release
 8  RUN yum -y update
 9
10  # OpenMPI Installation
11  RUN yum install -y which tar perl gcc gcc-c++ make
12  RUN curl -O https://www.open-mpi.org/software/ompi/v2.1/downloads/openmpi-2.1.1.tar.gz \
13      && gunzip -c openmpi-2.1.1.tar.gz | tar xf - \
14      && cd openmpi-2.1.1 \
```

```
15      && ./configure --prefix=/usr/local \
16      && make all install
17  RUN rm -r -f openmpi-2.1.1.tar.gz openmpi-2.1.1
18
19  ENV PATH /usr/local/bin:$PATH
20  ENV LD_LIBRARY_PATH /usr/local/lib:$LD_LIBRARY_PATH
21
22  # OpenCL Setting
23  RUN yum install -y ocl-icd clinfo
24
25  RUN mkdir -p /etc/OpenCL/vendors && \
26      echo "libnvidia-opencl.so.1" > /etc/OpenCL/vendors/nvidia.icd
27
28  RUN echo "/usr/local/nvidia/lib" >> /etc/ld.so.conf.d/nvidia.conf && \
29      echo "/usr/local/nvidia/lib64" >> /etc/ld.so.conf.d/nvidia.conf
30
31  ENV PATH /usr/local/nvidia/bin:${PATH}
32  ENV LD_LIBRARY_PATH /usr/local/nvidia/lib:/usr/local/nvidia/lib64:$LD_LIBRARY_PATH
33
34  # OpenSSH Setting (Public Key registeration)
35  RUN yum install -y openssh-server openssh-clients
36  RUN ssh-keygen -A
37  RUN ssh-keygen -f $HOME/.ssh/id_rsa -t rsa -N ''
38  RUN cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
39  RUN echo "StrictHostKeyChecking no" >> ~/.ssh/config
40
41  # Slurm Installation
42  RUN groupadd -r slurm && useradd -r -g slurm slurm
43
44  RUN yum -y install wget bzip2 perl gcc make munge munge-devel supervisor
45  RUN mkdir nfs && cd /nfs \
46      && wget http://www.schedmd.com/download/latest/slurm-17.02.6.tar.bz2
47  RUN cd /nfs && rpmbuild -ta slurm-17.02.6.tar.bz2
48  RUN cd /root/rpmbuild/RPMS/x86_64 \
49      && yum -y --nogpgcheck localinstall *
50
51  RUN set -x  \
52      && cd \
53      && rm -rf /usr/local/src/slurm \
54      && mkdir /etc/sysconfig/slurm \
55      && mkdir /var/spool/slurmd \
56      && chown slurm:slurm /var/spool/slurmd \
57      && mkdir /var/run/slurmd \
58      && chown slurm:slurm /var/run/slurmd \
59      && mkdir /var/lib/slurmd \
60      && chown slurm:slurm /var/lib/slurmd \
61      && /sbin/create-munge-key
62  COPY slurm.conf /etc/slurm/slurm.conf
63  COPY supervisord.conf /etc/
64
65  RUN yum -y install net-tools nano
66  RUN yum -y install ocl-icd-devel java
67
68  # Ganglia Install
69  RUN yum install -y ganglia-gmond
```

```
70
71 # RUN yum -y install mesa-\*
```

우선 위의 dockerfile 파일을 만들어놓고 slurm.conf, supervisord.conf, gmond.conf 파일을 만든 뒤 docker build를 하면 된다.

- OpenCL이 설치가 잘 안된 것 같으면 RUN yum -y install mesa-* 부분을 풀어 라이브러리를 더 설치한다.
- Dockerfile 에서 ENTRYPOINT는 CMD와 다르게 사용되어야 한다.

3.2.2. Create Slurm.conf

Configuration 파일을 만드는 URL은 다음과 같다.

- https://slurm.schedmd.com/configurator.html
- https://slurm.schedmd.com/configurator.easy.html

slurm.conf

```
 1 # slurm.conf
 2 #
 3 # See the slurm.conf man page for more information.
 4 #
 5 ClusterName=linux
 6 ControlMachine=n6
 7 #ControlAddr=
 8 #BackupController=
 9 #BackupAddr=
10 #
11 SlurmUser=slurm
12 #SlurmdUser=root
13 SlurmctldPort=6817
14 SlurmdPort=6818
15 AuthType=auth/munge
16 #JobCredentialPrivateKey=
17 #JobCredentialPublicCertificate=
18 StateSaveLocation=/var/lib/slurmd
19 SlurmdSpoolDir=/var/spool/slurmd
20 SwitchType=switch/none
21 MpiDefault=none
22 SlurmctldPidFile=/var/run/slurmd/slurmctld.pid
23 SlurmdPidFile=/var/run/slurmd/slurmd.pid
24 ProctrackType=proctrack/pgid
25 #PluginDir=
26 CacheGroups=0
27 #FirstJobId=
28 ReturnToService=0
29 #MaxJobCount=
30 #PlugStackConfig=
31 #PropagatePrioProcess=
32 #PropagateResourceLimits=
33 #PropagateResourceLimitsExcept=
34 #Prolog=
35 #Epilog=
```

```
#SrunProlog=
#SrunEpilog=
#TaskProlog=
#TaskEpilog=
#TaskPlugin=
#TrackWCKey=no
#TreeWidth=50
#TmpFS=
#UsePAM=
#
# TIMERS
SlurmctldTimeout=300
SlurmdTimeout=300
InactiveLimit=0
MinJobAge=300
KillWait=30
Waittime=0
#
# SCHEDULING
SchedulerType=sched/backfill
#SchedulerAuth=
#SchedulerPort=
#SchedulerRootFilter=
SelectType=select/cons_res
SelectTypeParameters=CR_CPU_Memory
FastSchedule=1
#PriorityType=priority/multifactor
#PriorityDecayHalfLife=14-0
#PriorityUsageResetPeriod=14-0
#PriorityWeightFairshare=100000
#PriorityWeightAge=1000
#PriorityWeightPartition=10000
#PriorityWeightJobSize=1000
#PriorityMaxAge=1-0
#
# LOGGING
SlurmctldDebug=3
#SlurmctldLogFile=
SlurmdDebug=3
#SlurmdLogFile=
JobCompType=jobcomp/none
#JobCompLoc=
#
# ACCOUNTING
#JobAcctGatherType=jobacct_gather/linux
#JobAcctGatherFrequency=30
#
#AccountingStorageType=accounting_storage/slurmdbd
#AccountingStorageHost=
#AccountingStorageLoc=
#AccountingStoragePass=
#AccountingStorageUser=
#
# COMPUTE NODES
NodeName=c1 NodeHostName=n1 NodeAddr=172.17.0.2 CPUs=8 CoresPerSocket=4
```

```
   ThreadsPerCore=2 RealMemory=7785 TmpDisk=10229
91 NodeName=c2 NodeHostName=n2 NodeAddr=172.17.0.3 CPUs=8 CoresPerSocket=4
   ThreadsPerCore=2 RealMemory=7785 TmpDisk=10229
92 NodeName=c3 NodeHostName=n3 NodeAddr=172.17.0.4 CPUs=8 CoresPerSocket=4
   ThreadsPerCore=2 RealMemory=7785 TmpDisk=10229
93 NodeName=c4 NodeHostName=n4 NodeAddr=172.17.0.5 CPUs=8 CoresPerSocket=4
   ThreadsPerCore=2 RealMemory=7785 TmpDisk=10229
94 NodeName=c5 NodeHostName=n5 NodeAddr=172.17.0.6 CPUs=8 CoresPerSocket=4
   ThreadsPerCore=2 RealMemory=7785 TmpDisk=10229
95 #
96 # PARTITIONS
97 PartitionName=party Default=yes Nodes=c[1-5] Priority=50 DefMemPerCPU=500 Shared=NO
   MaxNodes=5 MaxTime=5-00:00:00 DefaultTime=5-00:00:00 State=UP
```

특히 COMPUTE NODES에서 각각에 HostName 과 NodeAddr를 설정해 주는 것이 중요하며,

```
$ slurmd -C
```

명령어로 얻을 수 있는 리소스 결과 (코어 개수, 메모리 용량 등)을 참고하여 설정한다.

3.2.3. Create supervisord.conf

munge와 slurmd를 실행하는 configuration. docker 에서 systemctl이 동작하지 않는 부분을 이 부분으로 처리한다.

supervisord.conf

```
 1 [unix_http_server]
 2 file=/var/run/supervisor/supervisor.sock
 3
 4 [supervisord]
 5 logfile=/var/log/supervisor/supervisord.log
 6 logfile_maxbytes=5MB
 7 logfile_backups=10
 8 loglevel=info
 9 pidfile=/var/run/supervisord.pid
10 nodaemon=false
11
12 [rpcinterface:supervisor]
13 supervisor.rpcinterface_factory = supervisor.rpcinterface:make_main_rpcinterface
14
15 [supervisorctl]
16 serverurl=unix:///var/run/supervisor/supervisor.sock
17
18 [program:munged]
19 user=munge
20 command=/usr/sbin/munged -F
21 autostart=true
22 autorestart=false
23 startsecs=5
24 startretries=2
25 exitcodes=0,1,2
26 stdout_logfile=/var/log/supervisor/munged.log
27 stdout_logfile_maxbytes=1MB
```

```
28 stdout_logfile_backups=5
29 stderr_logfile=/var/log/supervisor/munged.log
30 stderr_logfile_maxbytes=1MB
31 stderr_logfile_backups=5
32
33 [program:slurmctld]
34 user=root
35 command=/usr/sbin/slurmctld -D -vvvvv
36 autostart=true
37 autorestart=false
38 startsecs=5
39 startretries=2
40 exitcodes=0,1,2
41 stdout_logfile=/var/log/supervisor/slurmctld.log
42 stdout_logfile_maxbytes=1MB
43 stdout_logfile_backups=5
44 stderr_logfile=/var/log/supervisor/slurmctld.log
45 stderr_logfile_maxbytes=1MB
46 stderr_logfile_backups=5
47
48 [program:slurmd]
49 user=root
50 command=/usr/sbin/slurmd -D -vvvvv
51 autostart=true
52 autorestart=false
53 startsecs=5
54 startretries=2
55 exitcodes=0,1,2
56 stdout_logfile=/var/log/supervisor/slurmd.log
57 stdout_logfile_maxbytes=1MB
58 stdout_logfile_backups=5
59 stderr_logfile=/var/log/supervisor/slurmd.log
60 stderr_logfile_maxbytes=1MB
61 stderr_logfile_backups=5
```

3.2.4. Build Docker image

마지막으로 아래의 커맨드를 입력하여 최종적으로 빌드하면 된다.

```
$ docker build -t slurm-light .
```

해당 이미지가 저장되었음을 확인 할 수 있다.

```
$ docker images
slurm-light          latest          a4b85dec2c78          20 minutes ago          984.8 MB
```

3.2.5. Running Script

slurmscript

```
1 setenforce 0
2 sestatus
3 nvidia-docker run -d -h n1 --name c1 \
4 --add-host n1:172.17.0.2 \
```

```
 5  --add-host n2:172.17.0.3 \
 6  --add-host n3:172.17.0.4 \
 7  --add-host n4:172.17.0.5 \
 8  --add-host n5:172.17.0.6 \
 9  --add-host n6:172.17.0.7 \
10  -v /dockerdata:/dockerdata slurm-light /bin/bash -c \
11  "/usr/bin/supervisord -c /etc/supervisord.conf && gmond && /usr/sbin/sshd -D"
12
13  nvidia-docker run -d -h n2 --name c2 \
14  --add-host n1:172.17.0.2 \
15  --add-host n2:172.17.0.3 \
16  --add-host n3:172.17.0.4 \
17  --add-host n4:172.17.0.5 \
18  --add-host n5:172.17.0.6 \
19  --add-host n6:172.17.0.7 \
20  -v /dockerdata:/dockerdata slurm-light /bin/bash -c \
21  "/usr/bin/supervisord -c /etc/supervisord.conf && gmond && /usr/sbin/sshd -D"
22
23  nvidia-docker run -d -h n3 --name c3 \
24  --add-host n1:172.17.0.2 \
25  --add-host n2:172.17.0.3 \
26  --add-host n3:172.17.0.4 \
27  --add-host n4:172.17.0.5 \
28  --add-host n5:172.17.0.6 \
29  --add-host n6:172.17.0.7 \
30  -v /dockerdata:/dockerdata slurm-light /bin/bash -c \
31  "/usr/bin/supervisord -c /etc/supervisord.conf && gmond && /usr/sbin/sshd -D"
32
33  nvidia-docker run -d -h n4 --name c4 \
34  --add-host n1:172.17.0.2 \
35  --add-host n2:172.17.0.3 \
36  --add-host n3:172.17.0.4 \
37  --add-host n4:172.17.0.5 \
38  --add-host n5:172.17.0.6 \
39  --add-host n6:172.17.0.7 \
40  -v /dockerdata:/dockerdata slurm-light /bin/bash -c \
41  "/usr/bin/supervisord -c /etc/supervisord.conf && gmond && /usr/sbin/sshd -D"
42
43  nvidia-docker run -d -h n5 --name c5 \
44  --add-host n1:172.17.0.2 \
45  --add-host n2:172.17.0.3 \
46  --add-host n3:172.17.0.4 \
47  --add-host n4:172.17.0.5 \
48  --add-host n5:172.17.0.6 \
49  --add-host n6:172.17.0.7 \
50  -v /dockerdata:/dockerdata slurm-light /bin/bash -c \
51  "/usr/bin/supervisord -c /etc/supervisord.conf && gmond && /usr/sbin/sshd -D"
52
53  docker inspect -f "{{ .NetworkSettings.IPAddress }}" c1
54  docker inspect -f "{{ .NetworkSettings.IPAddress }}" c2
55  docker inspect -f "{{ .NetworkSettings.IPAddress }}" c3
56  docker inspect -f "{{ .NetworkSettings.IPAddress }}" c4
57  docker inspect -f "{{ .NetworkSettings.IPAddress }}" c5
58  nvidia-docker run -it --rm -h n6 --name c6 \
59  --add-host n1:172.17.0.2 \
```

```
--add-host n2:172.17.0.3 \
--add-host n3:172.17.0.4 \
--add-host n4:172.17.0.5 \
--add-host n5:172.17.0.6 \
--add-host n6:172.17.0.7 \
 -v /dockerdata:/dockerdata slurm-tau /bin/bash -c \
"/usr/bin/supervisord -c /etc/supervisord.conf && gmond && /bin/bash"

docker rm c1 -f
docker rm c2 -f
docker rm c3 -f
docker rm c4 -f
docker rm c5 -f
```

이후 아래의 명령어로 한 번에 실행시킬 수 있다.

```
./slurmscript
```

3.3. Results

3.3.1. sinfo

```
[root@n6 /]# sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
normal*      up 5-00:00:00      5   idle c[1-5]
```

- **idle** 은 node가 연결되었으나 아무런 작업을 하지 않고 있음을 의미한다.
- **mix** 는 사용되고 있거나 다른 상태임을 의미한다.
- **unk** 노드가 연결되지 않았음을 의미한다.(unlinked)

asterisk (*) 표시는 해당 Partition이 default임을 의미한다.

3.3.2. squeue

squeue 명령어를 통해 현재 내가 진행한 job이 어떻게 진행되고 있는지도 알 수 있다.

```
[root@n6 sltest]# sbatch -w c1,c3,c5 -x c2 stress.slurm &
[1] 13194
[root@n6 sltest]# Submitted batch job 15
[1]+  Done                    sbatch -w c1,c3,c5 -x c2 stress.slurm
[root@n6 sltest]# squeue
             JOBID PARTITION     NAME  USER ST       TIME  NODES NODELIST(REASON)
                15       debug stress.s          root  R            0:03     5 c[1,3-6]
```

참고: 계산시간을 주는 위의 stress.slurm 파일은 다음과 같으며 yum install stress로 설치 해 주어야 한다.

stress.slurm

```
#!/bin/bash
stress -c 8 -i 1 -m 1 -t 10s
hostname
```

결과값은 10초 뒤에 hostname이 찍혀서 나온다.

3.3.3. scontrol show nodes

```
[root@n6 /]# scontrol show nodes
NodeName=c1 CoresPerSocket=4
   CPUAlloc=0 CPUErr=0 CPUTot=8 CPULoad=N/A
   AvailableFeatures=(null)
   ActiveFeatures=(null)
   Gres=(null)
   NodeAddr=172.17.0.2 NodeHostName=n1 Port=0
   RealMemory=7785 AllocMem=0 FreeMem=N/A Sockets=1 Boards=1
   State=IDLE ThreadsPerCore=2 TmpDisk=10229 Weight=1 Owner=N/A MCS_label=N/A
   Partitions=normal
   BootTime=2017-07-17T00:12:05 SlurmdStartTime=2017-07-17T00:32:40
   CfgTRES=cpu=8,mem=7785M
   AllocTRES=
   CapWatts=n/a
   CurrentWatts=0 LowestJoules=0 ConsumedJoules=0
   ExtSensorsJoules=n/s ExtSensorsWatts=0 ExtSensorsTemp=n/s

NodeName=c2 CoresPerSocket=4
   CPUAlloc=0 CPUErr=0 CPUTot=8 CPULoad=N/A
   AvailableFeatures=(null)
   ActiveFeatures=(null)
   Gres=(null)
   NodeAddr=172.17.0.3 NodeHostName=n2 Port=0
   RealMemory=7785 AllocMem=0 FreeMem=N/A Sockets=1 Boards=1
   State=IDLE ThreadsPerCore=2 TmpDisk=10229 Weight=1 Owner=N/A MCS_label=N/A
   Partitions=normal
   BootTime=2017-07-17T00:12:05 SlurmdStartTime=2017-07-17T00:32:40
   CfgTRES=cpu=8,mem=7785M
   AllocTRES=
   CapWatts=n/a
   CurrentWatts=0 LowestJoules=0 ConsumedJoules=0
   ExtSensorsJoules=n/s ExtSensorsWatts=0 ExtSensorsTemp=n/s

… c3, c4, c5 similar
```

여기서 확인 할 수 있듯이 NodeAddr=172.17.0.2~6 가 slurm.conf 에서 연결 되어 있어야 한다.

3.3.4. srun -N5 hostname

```
[root@n6 /]# srun -N5 hostname
n4
n1
n2
n5
n3
```

3.3.5. srun -n 5 a.out

```
[root@n6 smpi]# srun -n5 a.out
1 processes is alive
2000000000  3.1415926535898 n, rslt0
2000000001 ncall0
        43.117 s        0.71861 m       0.011977 h      0.00049904 d
1 processes is alive
2000000000  3.1415926535898 n, rslt0
2000000001 ncall0
        58.515 s        0.97525 m       0.016254 h      0.00067725 d
1 processes is alive
2000000000  3.1415926535898 n, rslt0
2000000001 ncall0
        58.621 s        0.97702 m       0.016284 h      0.00067849 d
1 processes is alive
2000000000  3.1415926535898 n, rslt0
2000000001 ncall0
        63.385 s        1.0564 m        0.017607 h      0.00073363 d
1 processes is alive
2000000000  3.1415926535898 n, rslt0
2000000001 ncall0
        63.517 s        1.0586 m        0.017644 h      0.00073515 d
```

이 경우, MPI 계산을 위해 5개의 Core를 할당하기 보다는 5개의 job을 실행하였음을 볼 수 있다.

3.3.6. salloc -n 5 mpirun --allow-run-as-root a.out

```
[root@n6 smpi]# salloc -n 5 mpirun --allow-run-as-root a.out
salloc: Granted job allocation 9
5 processes are alive
2000000000  3.1415926535897 n, rslt0
2000000001 ncall0
      9.4052 s      0.15675 m     0.0026126 h     0.00010886 d
salloc: Relinquishing job allocation 9
```

Salloc 명령어를 사용하면 원하는 수의 core를 할당하여 병렬화된 계산을 할 수 있다.

3.3.7. salloc -n40 mpirun --allow-run-as-root example1

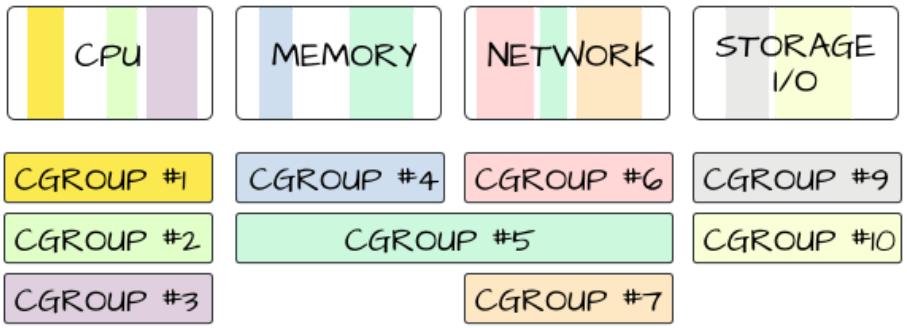
현재 Slurm.conf 에 등록된 node는 5개 이므로 8 x 5 = 40개의 Core를 할당할 수 있다. 이것을 출력 해 보면 아래와 같이 노드별로 core가 무작위적으로 할당되어 계산된 것을 볼 수 있다.

```
[root@n6 example1]# salloc -n40 mpirun --allow-run-as-root example1
salloc: Granted job allocation 9
Hello world from processor n2, rank 10 out of 40 processors
Hello world from processor n5, rank 32 out of 40 processors
Hello world from processor n2, rank 11 out of 40 processors
Hello world from processor n5, rank 37 out of 40 processors
Hello world from processor n1, rank 0 out of 40 processors
```

```
Hello world from processor n2, rank 8 out of 40 processors
Hello world from processor n2, rank 12 out of 40 processors
Hello world from processor n1, rank 1 out of 40 processors
Hello world from processor n2, rank 13 out of 40 processors
Hello world from processor n5, rank 38 out of 40 processors
Hello world from processor n1, rank 3 out of 40 processors
Hello world from processor n2, rank 15 out of 40 processors
Hello world from processor n5, rank 33 out of 40 processors
Hello world from processor n1, rank 6 out of 40 processors
Hello world from processor n2, rank 9 out of 40 processors
Hello world from processor n5, rank 35 out of 40 processors
Hello world from processor n1, rank 4 out of 40 processors
Hello world from processor n2, rank 14 out of 40 processors
Hello world from processor n1, rank 5 out of 40 processors
Hello world from processor n5, rank 36 out of 40 processors
Hello world from processor n5, rank 34 out of 40 processors
Hello world from processor n4, rank 26 out of 40 processors
Hello world from processor n1, rank 2 out of 40 processors
Hello world from processor n3, rank 23 out of 40 processors
Hello world from processor n3, rank 22 out of 40 processors
Hello world from processor n1, rank 7 out of 40 processors
Hello world from processor n3, rank 17 out of 40 processors
Hello world from processor n5, rank 39 out of 40 processors
Hello world from processor n4, rank 28 out of 40 processors
Hello world from processor n4, rank 31 out of 40 processors
Hello world from processor n3, rank 21 out of 40 processors
Hello world from processor n3, rank 18 out of 40 processors
Hello world from processor n3, rank 16 out of 40 processors
Hello world from processor n4, rank 25 out of 40 processors
Hello world from processor n3, rank 19 out of 40 processors
Hello world from processor n4, rank 29 out of 40 processors
Hello world from processor n3, rank 20 out of 40 processors
Hello world from processor n4, rank 30 out of 40 processors
Hello world from processor n4, rank 24 out of 40 processors
Hello world from processor n4, rank 27 out of 40 processors
salloc: Relinquishing job allocation 9
```

4. Cgroup

4.1. Overview



cgroup은 control group의 약자로, Linux Kernel 2.6.24 버전 이후부터 지원되며 시스템 상에서 동작 중인 태스크들을 임의로 그룹지어 제어할 수 있도록 도와주는 기능이다. "Everything is a file"이라는 unix의 paradigm에 맞게 cgroup은 파일시스템이며 마운트하여 사용된다.

- mount -t tmpfs cgroup_root /sys/fs/cgroup
- mkdir /sys/fs/cgroup/cpuset
- mount -t cgroup -ocpuset cpuset /sys/fs/cgroup/cpuset

또한 위의 그림처럼 CPU, Memory, Network, Storage I/O 등을 임의로 그룹지어진 Cgroup을 단위로 리소스를 분배하고 제한을 둘 수 있다.

Cgroup 디렉토리(/sys/fs/cgroup)에는 아래의 서브디렉토리들이 있는데 각각이 담당하는 바는 아래와 같다. (출처: https://access.redhat.com/documentation/ko-KR/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html)

- blkio — 이 서브시스템은 물리 드라이브 (예: 디스크, 솔리드 스테이트, USB 등)와 같은 블록 장치에 대한 입력/출력 액세스에 제한을 설정합니다.
- cpu — 이 서브시스템은 CPU에 cgroup 작업 액세스를 제공하기 위해 스케줄러를 사용합니다.
- cpuacct — 이 하위 시스템은 cgroup의 작업에 사용된 CPU 자원에 대한 보고서를 자동으로 생성합니다.
- cpuset — 이 서브시스템은 개별 CPU (멀티코어 시스템에서) 및 메모리 노드를 cgroup의 작업에 할당합니다.
- devices — 이 서브시스템은 cgroup의 작업 단위로 장치에 대한 액세스를 허용하거나 거부합니다.
- freezer — 이 서브시스템은 cgroup의 작업을 일시 중지하거나 다시 시작합니다.
- memory — 이 서브시스템은 cgroup의 작업에서 사용되는 메모리에 대한 제한을 설정하고 이러한 작업에서 사용되는 메모리 자원에 대한 보고서를 자동으로 생성합니다.
- net_cls — 이 서브시스템은 Linux 트래픽 컨트롤러 (tc)가 특정 cgroup 작업에서 발생하는 패킷을 식별하게 하는 클래식 식별자 (classid)를 사용하여 네트워크 패킷에 태그를 지정합니다.
- ns — *namespace* 서브시스템

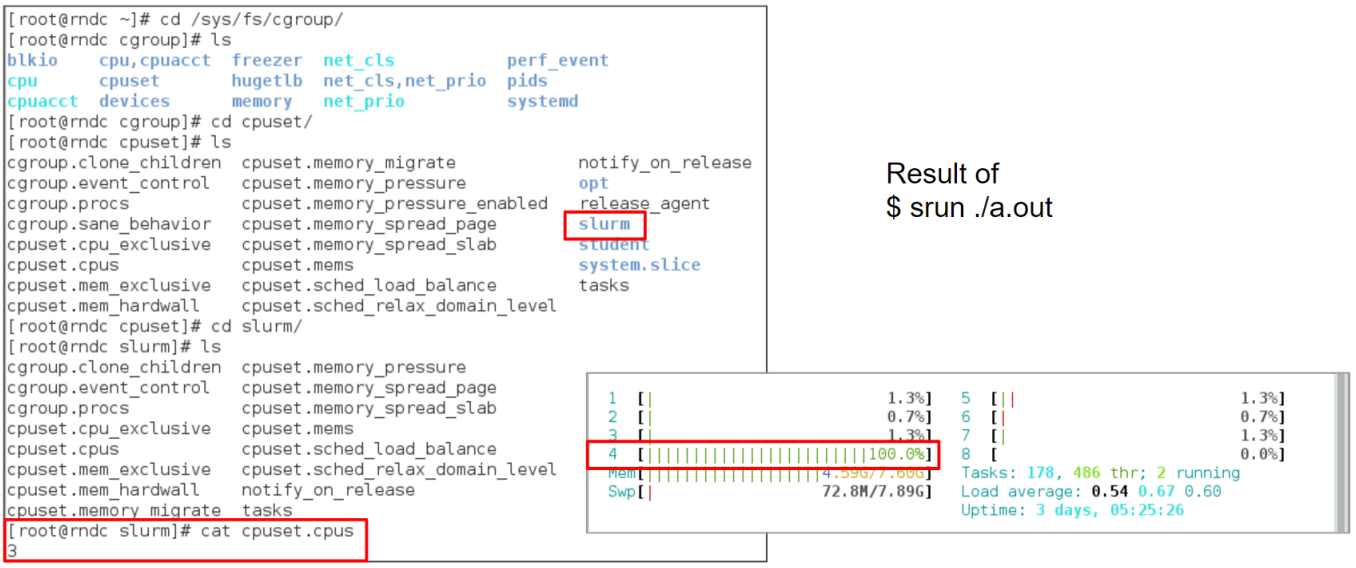
아래의 명령어들은 cgroup과 관계된 systemd 명령어 들이다.

- systemd-run(1) — The manual page lists all command-line options of the systemd-run utility.
- systemctl(1) — The manual page of the **systemctl** utility that lists available options and commands.
- systemd-cgls(1) — This manual page lists all command-line options of the systemd-cgls utility.
- systemd-cgtop(1) — The manual page contains the list of all command-line options of the systemd-cgtop utility.
- machinectl(1) — This manual page lists all command-line options of the machinectl utility.
- systemd.kill(5) — This manual page provides an overview of kill configuration options for system units.

출처: Radhat (https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html-single/Resource_Management_Guide/index.html)

자세한 명령어의 사용 방법은 핵심되는 내용은 아니기에 보고서에서 크게 다루지는 않았다.

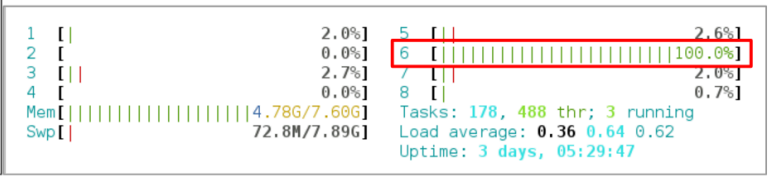
4.2. Usage Example



먼저 위 그림은 slurm이라는 cgroup의 cpuset.cpus가 3번, 다시 말해 4번째 core만 작업에 할당될 수 있도록 설정한 이후에 프로그램을 실행한 결과이다. 그 결과 4번 코어만 작업을 수행하는 동안 가동된 것을 확인할 수 있다.

```
[root@rndc slurm]# ls
cgroup.clone_children  cpuset.memory_pressure
cgroup.event_control   cpuset.memory_spread_page
cgroup.procs           cpuset.memory_spread_slab
cpuset.cpu_exclusive   cpuset.mems
cpuset.cpus            cpuset.sched_load_balance
cpuset.mem_exclusive   cpuset.sched_relax_domain_level
cpuset.mem_hardwall    notify_on_release
cpuset.memory_migrate  tasks
[root@rndc slurm]# cat cpuset.cpus
3
[root@rndc slurm]# echo 5 > cpuset.cpus
[root@rndc slurm]# cat cpuset.cpus
5
```

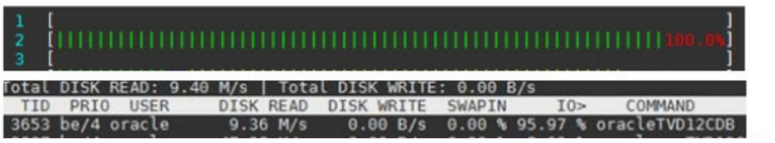
Result of
$ srun ./a.out



비슷한 방식으로 Memory 나 Disk I/O, Network I/O 에 대해서도 Limit을 걸 수 있다.

■ Requirements
– Usage of just one CPU
– Maximum of 10MB/s IO read und write

```
echo 1 > cpuset.cpus
echo "252:0 10485760" > blkio.throttle.read_bps_device
echo "252:0 10485760" > blkio.throttle.write_bps_device
```



여기서 “echo 5 > cpuset.cpus”로 cpuset.cpus파일에 5를 입력한다. 그리고 똑같이 slurm을 통해 프로세스를 실행한 결과 6번째 core만 사용되어 작업을 수행하였음을 볼 수 있다. 이처럼 파일시스템에서 파일에 입력하듯이 CPU, Memory, Disk, Network등을 모두 통제할 수 있으며 그 과정이 매우 간단하여 이를 기반으로 LXC나 Docker와 같은 기술들이 발전될 수 있었다.

4.3. Utilities

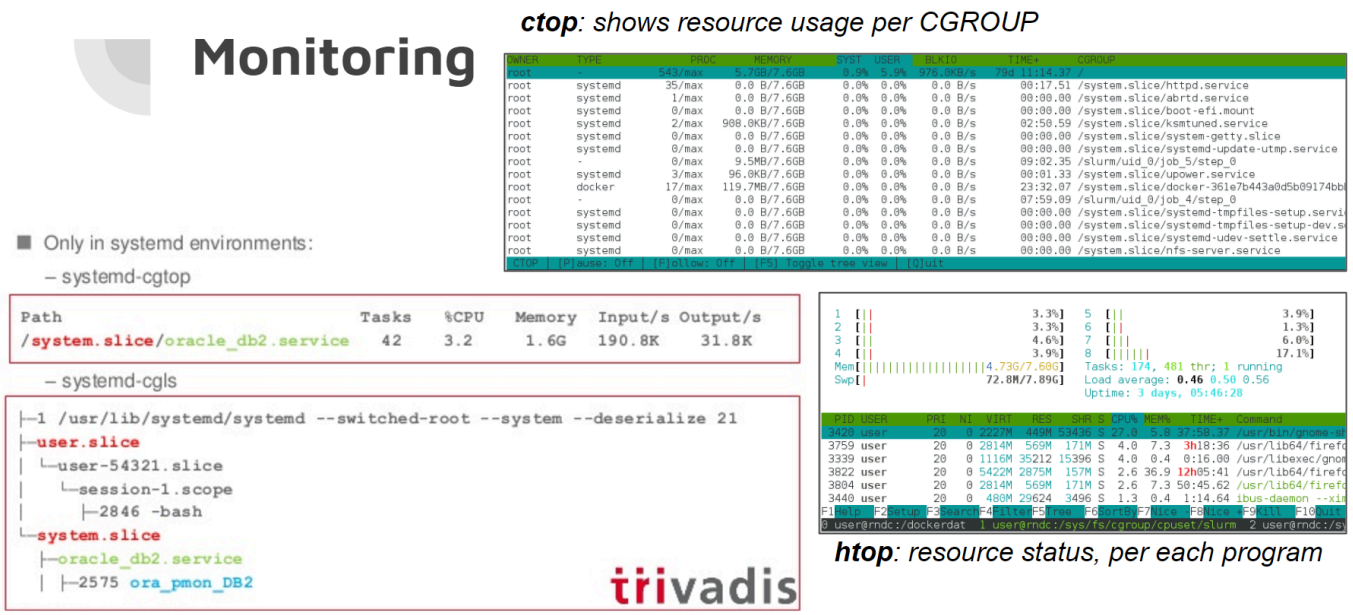
기본적으로 cgroup은 파일 시스템처럼 운용되기 때문에 직접 디렉토리를 만들어 cgroup에 마운트 하는 방식으로 사용할 수도 있다. 하지만 yum install libcgroup으로 cgroup 라이브러리를 설치한다면 좀 더 간편하게 명령어들을 사용하여 cgroup을 create 하거나 set 하는 등이 가능하다.

- Checking existing subsystems: lssubsys -am
- Configuration: cgcreate, cgset, cgsnapshot

4.4. Monitoring

systemd-cgtop: 각 slice에 속한 service 별로 CPU, Memory, Disk IO 에 대해 모니터링 해 주는 도구이다.

Path	Tasks	%CPU	Memory	Input/s	Output/s
/	227	29.5	5.7G	-	-
/system.slice	-	1.4	389.1M	-	-
/system.slice/gdm.service	2	1.2	2.9M	-	-
/system.slice/docker...f87ba1b6986a23648a69b2365176ca455.scope	6	0.1	156.9M	-	-
/system.slice/dbus.service	1	0.1	1.0M	-	-
/system.slice/gmond.service	1	0.0	7.4M	-	-
/system.slice/docker.service	3	0.0	52.1M	-	-
/system.slice/systemd-machined.service	1	0.0	632.0K	-	-
/system.slice/avahi-daemon.service	2	0.0	88.0K	-	-
/system.slice/httpd.service	8	0.0	-	-	-
/system.slice/systemd-logind.service	1	0.0	8.0K	-	-
/system.slice/tuned.service	1	0.0	268.0K	-	-
/system.slice/rngd.service	1	0.0	-	-	-
/system.slice/gssproxy.service	1	0.0	-	-	-
/system.slice/ModemManager.service	1	-	108.0K	-	-
/system.slice/NetworkManager.service	2	-	984.0K	-	-
/system.slice/abrt-oops.service	1	-	-	-	-



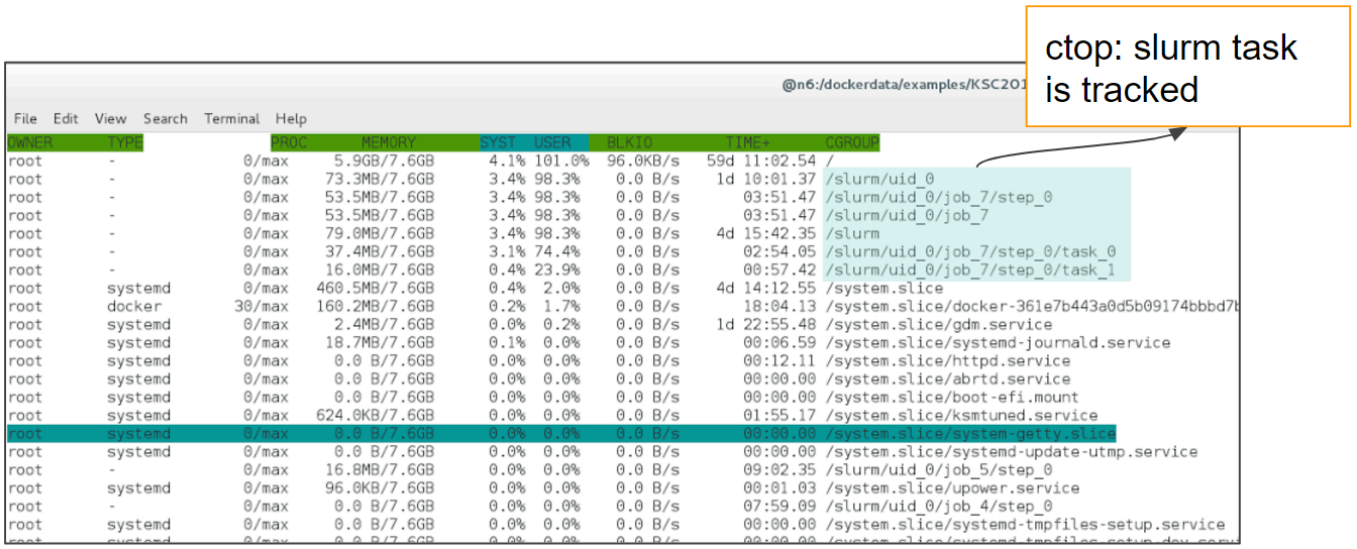
systemd-cgls: 각 slice 별로 프로세스의 pid를 출력하고 tree의 형태로 계층구조로 나타내어 보여준다.
ctop: CGROUP에 속한 프로세스들의 사용 량을 실시간으로 모니터링 하여 보여준다.
htop: 리소스가 사용되는 현황을 더 자세히 보여준다. (e.g. 각 CPU 별로 사용되는 현황 등)

4.5. Cgroup to Slurm

slurm.conf에서 아래의 내용을 수정 및 추가하면 cgroup을 사용할 수 있다.

slurm.conf
ProctrackType=proctrack/cgroup TaskPlugin=task/cgroup JobacctGatherType=jobacct_gather/cgroup

위의 ProctrackType은 process track type, TaskPlugin은 slurm에서의 task launch plugin, JobacctGatherType은 job accounting을 할 때의 mechanism의 종류를 의미하며 각각은 개별적으로 설정 가능하다. (반드시 cgroup으로 통일하여 사용되어야 하는 것은 아니다.) 또한 slurm으로 job을 실행하였을 경우에 slurm아래의 uid 아래의 job id로 tracking이 되어 모니터링이 가능하다.



단, docker에서 사용할 때는 마운트시 /sys/fs/cgroup 볼륨을 그대로 마운트 해 주어야 한다.

```
docker run -v /sys/fs/cgroup:/sys/fs/cgroup:ro
```

5. Program Profiling

5.1. Overview

HPC 환경에서는 각 노드들이 제대로 된 연산을 수행하고 있는지, 오류가 발생하지는 않았는지, 계산 효율은 얼마나 높아졌는지를 확인하기 위해 프로세스에 대한 프로파일링의 필요성이 생긴다. 다시말해 Process의 자원 사용량에 대한 분석이 필요하다. 특히 아래의 지표를 기준으로 프로파일링을 해 보고자 한다.

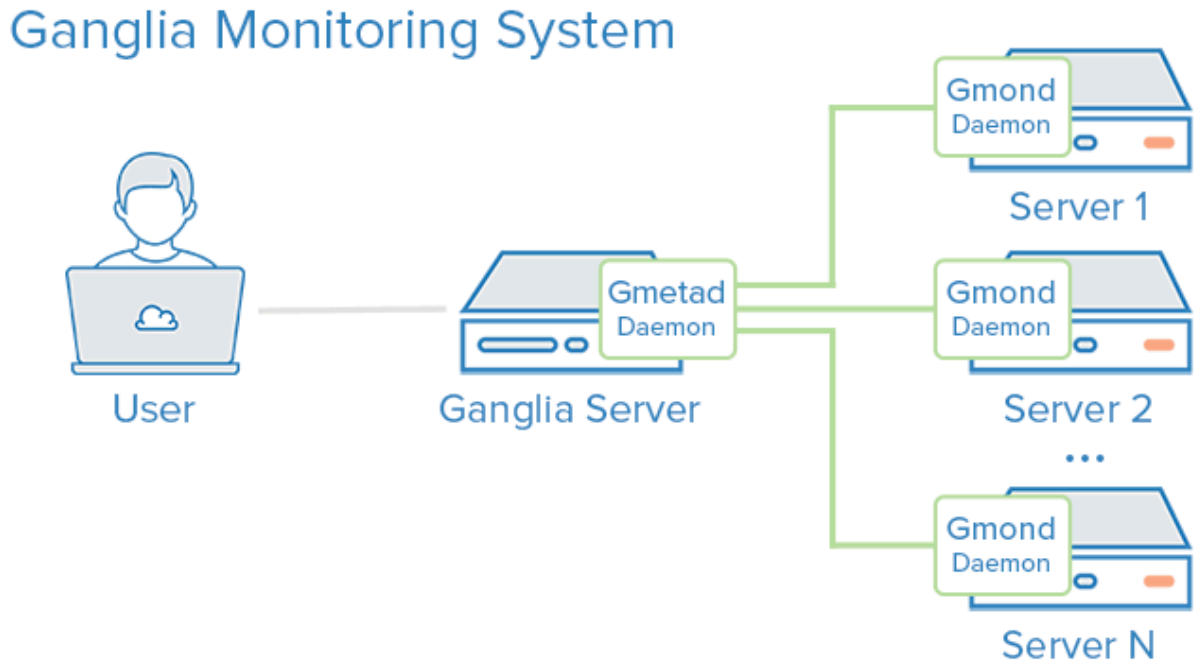
- CPU (사용 clock 수)
- Memory (Virtual memory 기준 사용량)
- Disk (Read/Write 데이터 양, IO 횟수)
- Network (Read/Write, IO 횟수, 통신 실패 횟수, 재시도 횟수)
- GPU 사용량 (Device 별로, 사용시간, Memory)

기본적으로 Process가 얼마나 리소스를 소모하는지를 확인하기위해 Linux 에서 자체적으로 제공하는 커맨드가 있긴 하지만 순간적인 지표이거나 원하는 형태로 분석된 결과를 얻기가 힘든 경우가 많다. 따라서 Profiling을 위한 Tool들을 설치할 필요성이 있다. 아래의 표는 이번 프로젝트에서 사용한 Profiling Tool들의 리스트를 소개하고 있으며 각각의 설치법과 사용 방법에 대해 소개 하고자한다.

Program	Main Feature	CPU	Memory	Disk	Network	GPU	비고
Ganglia	종합적으로 각 host의 resource를 관리 가능한 Monitoring System	✓	✓	✓	✓		
TAU (Tuning and Analysis Utility)	MPI에서 메시지가 Host끼리 어떻게 전송되었는지를 시계열로 나타내어줌	✓					
Valgrind	Memory Leaks 등을 확인 할 수 있는 메모리 관련 프로그램		✓				
WireShark	네트워크 통신을 시계열로 나타내어줌				✓		
OpenCL Event Profiling	OpenCL에서 기본적으로 Event변수를 이용해 Profiling 하는 방법					✓	clGetEventProfilingInfo
NVVP (NVIDIA Visual Profiler)	NVIDIA에서 CUDA를 Profiling하기 위해 만든 프로그램					✓	Only Supports NVIDIA CUDA

5.2. Ganglia

5.2.1. Ganglia Server Installation



Ganglia는 Gmetad가 돌아가는 Ganglia Server와 모니터링 하고자 하는 서버들에 각각 Gmond 프로그램을 설치하고 configuration 을 설정 해 주어야 한다. 또한 웹 앱의 형태로 모니터링을 실시간으로 나타내어 주기 때문에 관련된 프로그램을 설치해 주어야 한다.
우선 최신의 yum repository를 갖고있는 epel-release를 설치한다.

```
$ yum -y install epel-release && yum -y update
```

이후 웹 서비스와 관련된 PHP 계열의 프로그램을 설치해준다.

```
$ yum install httpd php php-mysql php-gd php-ldap php-odbc php-pear php-xml php-xmlrpc
php-mbstring php-snmp php-soap curl curl-devel
```

또한 ganglia server 프로그램도 설치해 준다.

```
$ yum install rrdtool rrdtool-devel ganglia-web ganglia-gmetad ganglia-gmond-python httpd apr-devel
zlib-devel libconfuse-devel expat-devel pcre-devel
```

이후 /etc/hosts 파일에 host ip 주소와 이름을 입력해 기존의 내용에 추가해준다. 예를 들면 아래와 같다. (현재 내 host 이름을 rndc라고 하자).

/etc/hosts

```
192.168.100.6 rndc
172.17.0.2 n1
172.17.0.3 n2
172.17.0.4 n3
```

다음 Firewall과 Selinux를 disable 시킨다.

```
$ systemctl stop firewalld
$ systemctl disable firewalld
$ vi /etc/sysconfig/selinux
```

→ SELINUX=disabled 부분으로 변경
그리고 /etc/ganglia/ 디렉토리에 있는 gmetad.conf와 gmond.conf 파일을 수정해준다.
수정하는 부분은 아래와 같다.

gmetad.conf

```
data_source "RNDC Monitoring" rndc
```

gmond.conf

```
cluster {
  name = "RNDC Monitoring"
  ...
}

udp_send_channel {
  # mcast_join = .... (주석처리)
  host = rndc # 추가된 부분
  port = 8649
  ttl = 1
}

udp_recv_channel {
  # mcast_join = .... (주석처리)
  port = 8649
  # bind ...
  # retry_bind ...
  # buffer ... (나머지 전부 주석처리)
}

tcp_accept_channel {
  port = 8649
  # gzip_output = no
}
```

다음 /etc/httpd/conf.d/ganglia.conf 를 수정해 준다.

/etc/httpd/conf.d/ganglia.conf

```
Alias /ganglia /usr/share/ganglia

<Location /ganglia>
  Require all granted
  # Require ip 10.1.2.3
```

```
  # Require host example.org
</Location>
```

최종적으로 httpd와 ganglia를 재시작 해준다.

```
# systemctl restart httpd
# systemctl enable httpd
# systemctl restart gmond
# systemctl restart gmetad
# systemctl enable gmond
# systemctl enable gmetad
```

이후에 웹페이지에서 http://rndc/ganglia를 접속해 현재 설치 상태를 확인한다.

5.2.2. Ganglia Client Installation

Client에서의 설치는 간단하다. 우선 /etc/hosts 파일을 수정하여 server 주소 (rndc)를 추가해준다. 이후,

```
$ yum -y install ganglia-gmond
```

로 ganglia monitoring 프로그램을 설치 하고, gmond.conf 파일을 수정한다.

/etc/ganglia/gmond.conf

```
cluster {
  name = "RNDC Monitoring"
  ...
}

udp_send_channel {
  host = rndc # 추가
  port = 8649 # 주석 삭제
  ttl = 1 # 주석 삭제
}

udp_recv_channel {
  # mcast_join = ...
  # port = ...
  # bind = ...
  # retry_bind = ...
  # buffer = ... # 전부 주석처리
}

tcp_accept_channel {
  # port = 8649
  # gzip_output = no # 주석처리
}
```

이후 gmond를 재시작 하여 사용한다.

5.2.3. Dockerfile과 실행script

위의 host를 설정하더라도 docker에서는 실행시 /etc/hosts의 호스트를 초기화 시키는 특징을 가지고 있다. 따라서 최초 실행시 호스트를 등록해 주어야 한다.
따라서 docker 실행시 아래와 같은 스크립트로 실행한다.

```
$ nvidia-docker run -d -h n1 --name c1 \
--add-host n1:172.17.0.2 \
--add-host n2:172.17.0.3 \
--add-host n3:172.17.0.4 \
--add-host n4:172.17.0.5 \
--add-host n5:172.17.0.6 \
--add-host n6:172.17.0.7 \
-v /dockerdata:/dockerdata slurm-light /bin/bash -c \
"/usr/bin/supervisord -c /etc/supervisord.conf && gmond && /usr/sbin/sshd -D"
```

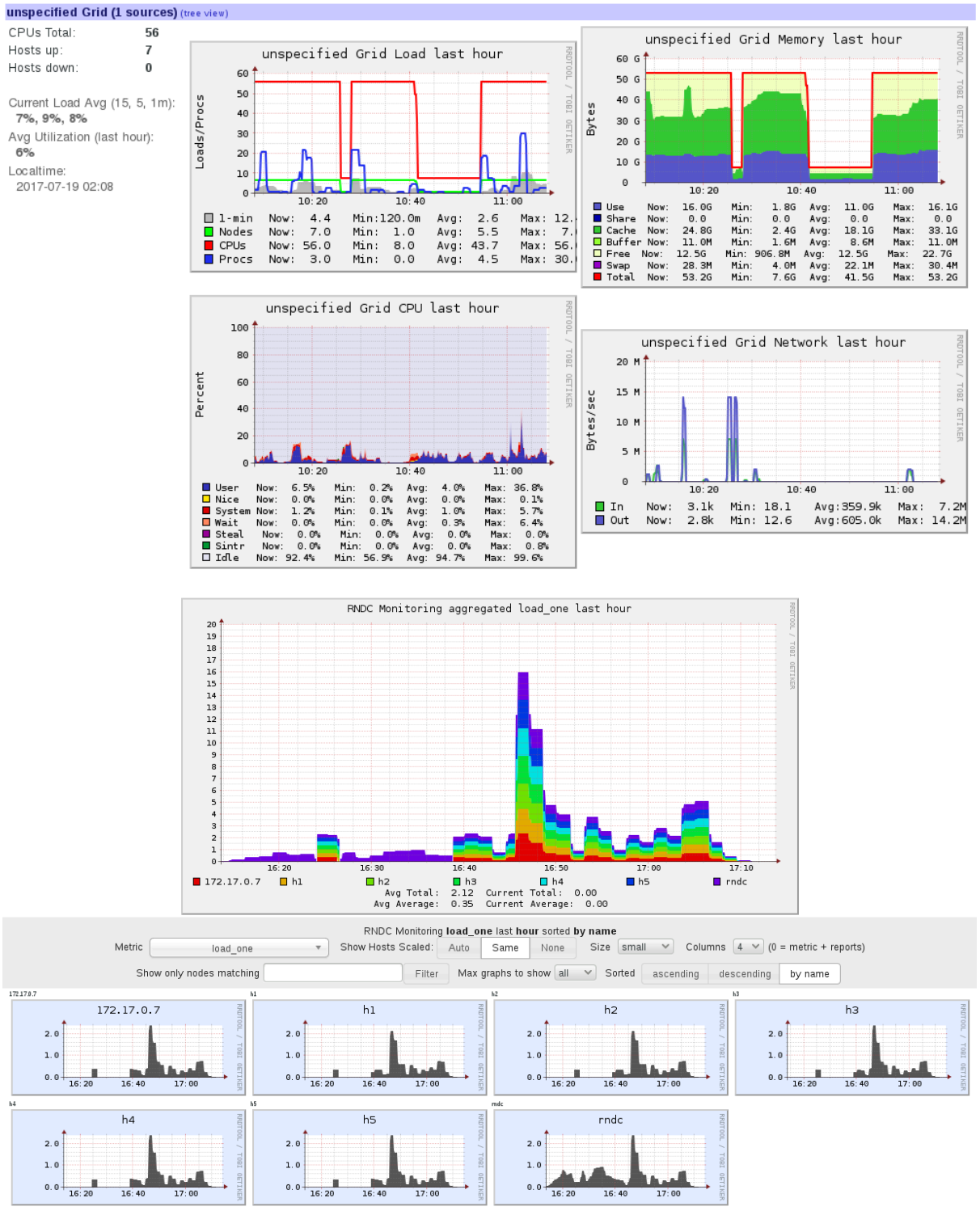
또한 dockerfile에서 설치 방법도 약간 차이가 있다. 위의 Docker에서 gmond setting 과정에서 발생한 gmond.conf 파일을 미리 생성 하여 놓고 설치 이후 COPY하는 식으로 하면 효율적이다.

Ganglia server 설치는 GUI가 지원 되는 메인 서버 컴퓨터에 설치를 하여 웹 페이지를 열어 확인 할 수 있게 설정한다. 반면 우리가 Dockerfile에서 설치하는 것은 ganglia-client 부분 만을 취한다. 아래의 내용을 dockerfile 에 추가해 주면 된다.

dockerfile (+)

```
# Ganglia Install
RUN yum install -y ganglia-gmond
COPY gmond.conf /etc/ganglia/gmond.conf
```

5.2.4. 설치 결과



5.3. Tuning and Analysis Utility (TAU)

5.3.1. Download

tau, pdt를 웹에서 다운로드 한다.

```
$ wget -c http://tau.uoregon.edu/tau.tgz
$ wget -c http://tau.uoregon.edu/pdt.tgz
```

```
$ tar zxf tau.tgz
$ tar zxf pdt.tgz
```

5.3.2. PDT 설치

-pdt 옵션에 쓰기 위한 과정이다.

```
$ cd pdtoolkit-x.xx
$ ./configure
$ make
$ make install
$ cd ..
```

5.3.3. TAU 설치

```
$ cd tau-x.xx.x
```

TAU를 본격적으로 설치하기 전에 아래의 directory 정보를 알아야 한다. 이 때 예전에 /usr/local에 openmpi를 설치했다는 것을 기억해야 된다.

- -mpiinc에 들어갈 openmpi include 정보: /usr/local/include
- -mpilib에 들어갈 openmpi library정보: /usr/local/lib

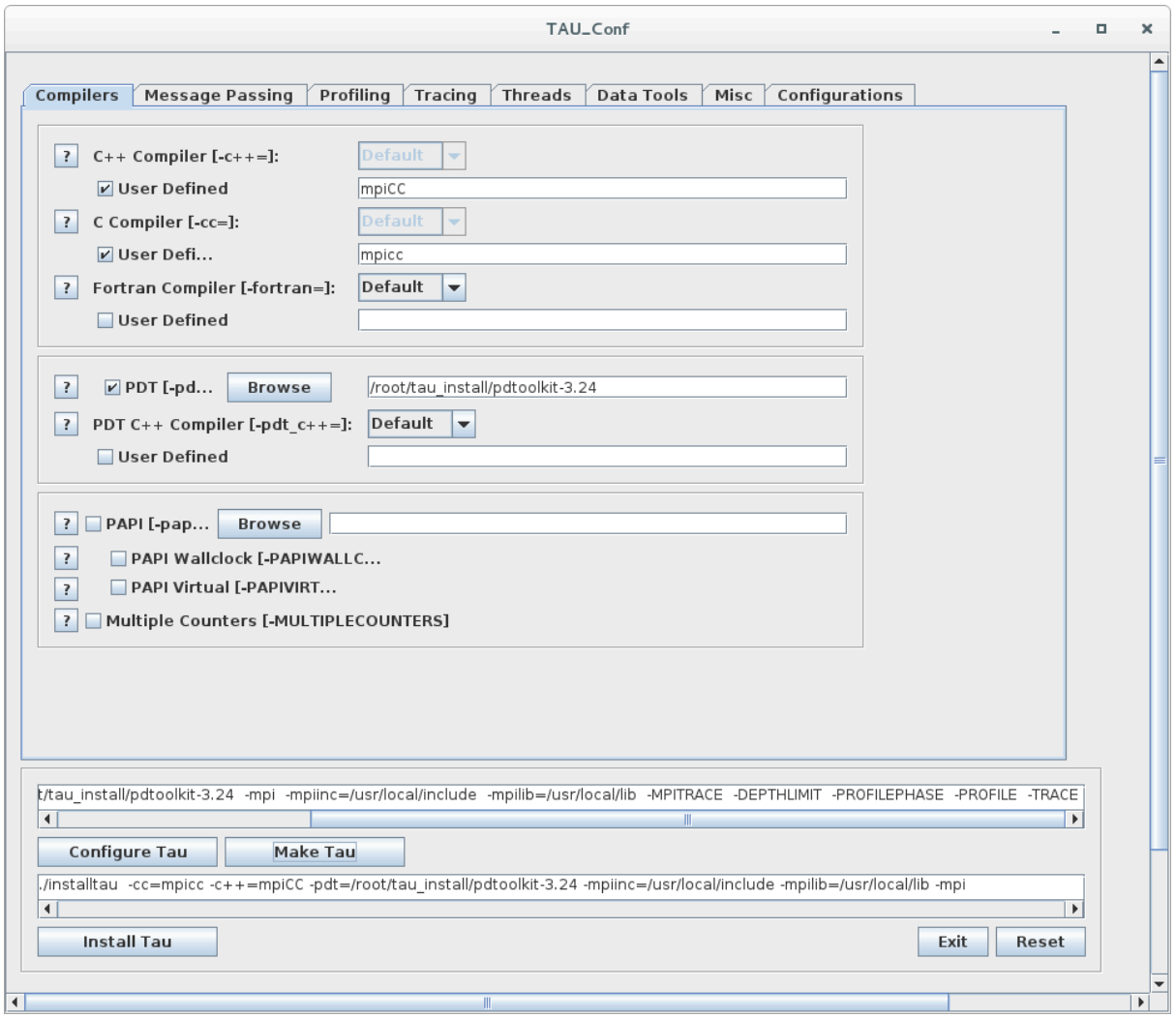
이제 아래와 같이 configure해준다. -c++=mpiCC -cc=mpicc 등등에 유의해서 적어준다.

```
$ ./configure -c++=mpiCC -cc=mpicc -mpi -MPITRACE -slog2 -PROFILE -TRACE -openmp
-pdt=/root/data/tau_install/pdtoolkit-3.24 -mpiinc=/usr/local/include -mpilib=/usr/local/lib -opencl="/usr/"
```

특히 pdt가 설치된 경로를 상대경로로 쓰면 절대 안된다. (너무 당연해 보이지만 이것 때문에 설치가 계속 안되서 헤메었다.) 그 외에 아래 처럼 추가적으로 다른 설정들을 더 넣어 설치 할 수 있다.

```
$ ./configure -c++=mpiCC  -cc=mpicc  -pdt=/root/tau_install/pdtoolkit-3.24  -mpi
-mpiinc=/usr/local/include  -mpilib=/usr/local/lib -openmp -opari  -MPITRACE  -DEPTHLIMIT
-PROFILEPHASE  -PROFILE  -TRACE
< 또는 >
$ ./configure -c++=mpiCC  -cc=mpicc  -pdt=/root/tau_install/pdtoolkit-3.24  -mpi
-mpiinc=/usr/local/include  -mpilib=/usr/local/lib  -MPITRACE  -PROFILEPHASE  -PROFILEMEMORY
-PROFILE  -TRACE  -slog2  -openmp  -CPUTIME  -LINUXTIMERS  -iowrapper
```

또한 이 방법 이외에 /tau_setup이라는 프로그램도 제공하는데, 이를 이용하면 GUI에서 configure 세팅을 해서 configure, make, make install 까지 버튼만으로 가능하게 설정해 두었다.



이후 make와 make install을 순차적으로 입력해준다.

```
$ make
<make log...>
$ make install
<make install log...>
```

5.3.4. PATH와 TAU_MAKEFILE 설정

bin이 들어있는 PATH와 TAU_MAKEFILE을 설정해 주어야 한다. 그래서 .bashrc파일 혹은 .bash_profile 파일을 수정해 주고 source 명령어로 반영 해 주어야 한다. path는 당연한 것이지만 tau_makefile은 설정해 주어야 tau_cc.sh 와 같은 명령어가 제대로 동작한다.

```
$ echo "PATH=$PATH:$HOME/bin:/root/tau_install/tau-2.26.2/x86_64/bin" >> ~/.bash_profile
$ echo
"TAU_MAKEFILE="/src/downloads/tau-2.26.2/x86_64/lib/Makefile.tau-memory-depthlimit-phase-mpi-pdt-profile-trace-mpitrace" >> ~/.bash_profile
$ source ~/.bash_profile
```

Makefile은 설정시 입력한 옵션에 따라 달라질 수 있다.

5.3.5. 예제 돌려보기

이전에 짰던 simpson 알고리즘을 기준으로 테스트 해보자.
우선 컴파일 자체를 tau_cc.sh로 해 주어야 한다.

```
$ tau_cc.sh simpson-mpicl.c -lOpenCL
```

-o로 따로 output file을 설정하지 않았으므로 a.out이 기본적으로 나온다.
이제 mpirun를 해주면…

```
$ mpirun --allow-run-as-root -n 4 a.out
```

그러면 tautrace.0.0.0.trc ~ tautrace.3.0.0.trc, events.0.edf ~ events.3.edf 와 같은 파일이 결과로 나온다.
그리고 tau_treemerge.pl을 입력하면

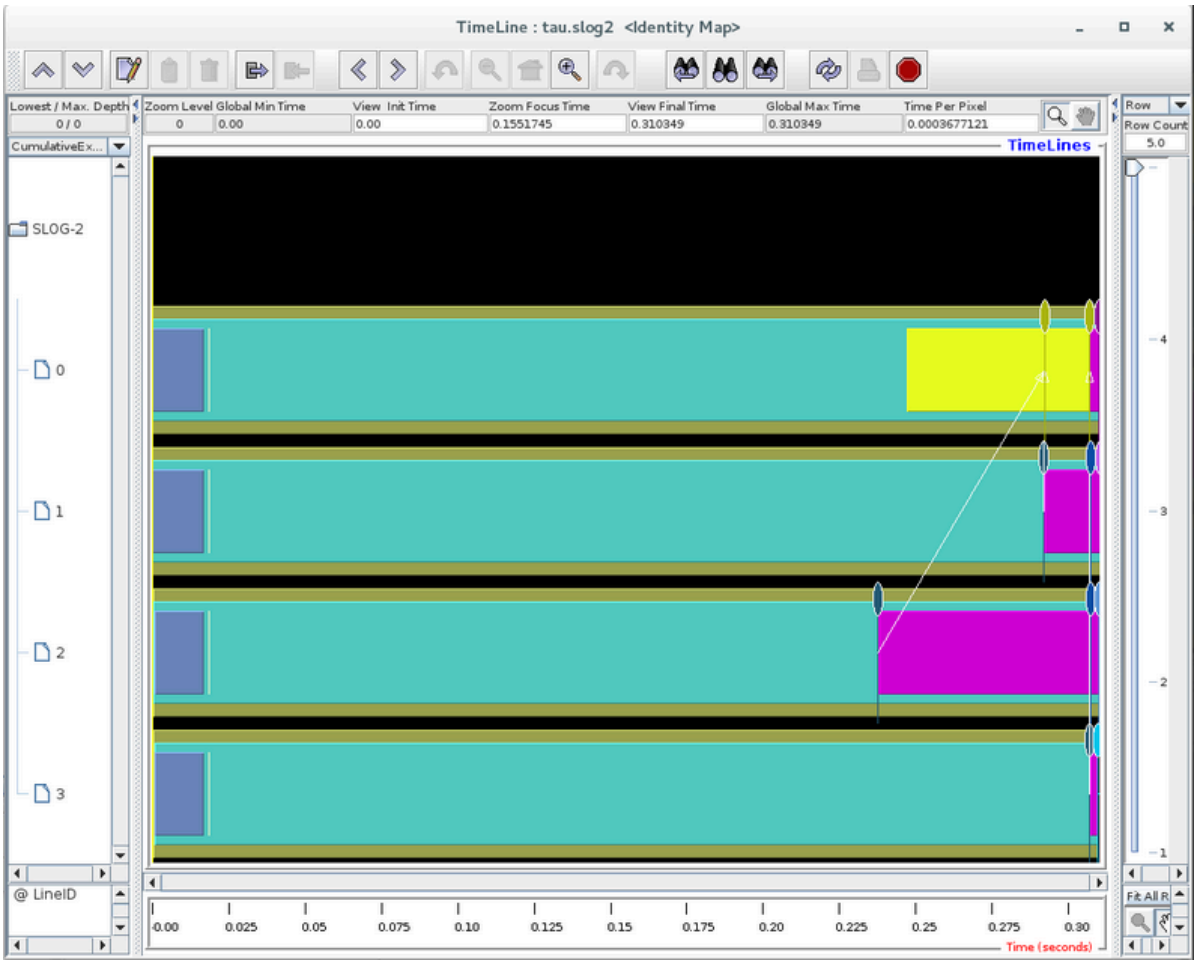
```
[root@localhost simpson]# tau_treemerge.pl
/home/user/Downloads/tau-2.26.2/x86_64/bin/tau_merge -m tau.edf -e events.0.edf events.1.edf
events.2.edf events.3.edf tautrace.0.0.0.trc tautrace.1.0.0.trc tautrace.2.0.0.trc tautrace.3.0.0.trc tau.trc
tautrace.0.0.0.trc: 258 records read.
tautrace.1.0.0.trc: 36 records read.
tautrace.2.0.0.trc: 36 records read.
tautrace.3.0.0.trc: 36 records read.
[root@localhost simpson]# ls
a.out         events.2.edf  simpson-mpicl.c   tau.edf            tautrace.2.0.0.trc
events.0.edf  events.3.edf  simpson-mpicl.cl  tautrace.0.0.0.trc  tautrace.3.0.0.trc
events.1.edf  mhf           simpson-mpicl.o   tautrace.1.0.0.trc  tau.trc
```

이처럼 tau.edf, tau.trc 파일이 발생한다. 이후

```
$ tau2slog2 tau.trc tau.edf -o tau.slog2
```

(Tip: tau2slog2만 입력해도 tau2slog2 tau.trc tau.edf -o tau.slog2와 같은 방식으로 입력하라고 안내 글이 나온다.)
를 입력해 주면 tau.slog2 파일이 나온다.
이를 이용해 jumpshot을 해주면

```
$ jumpshot
```

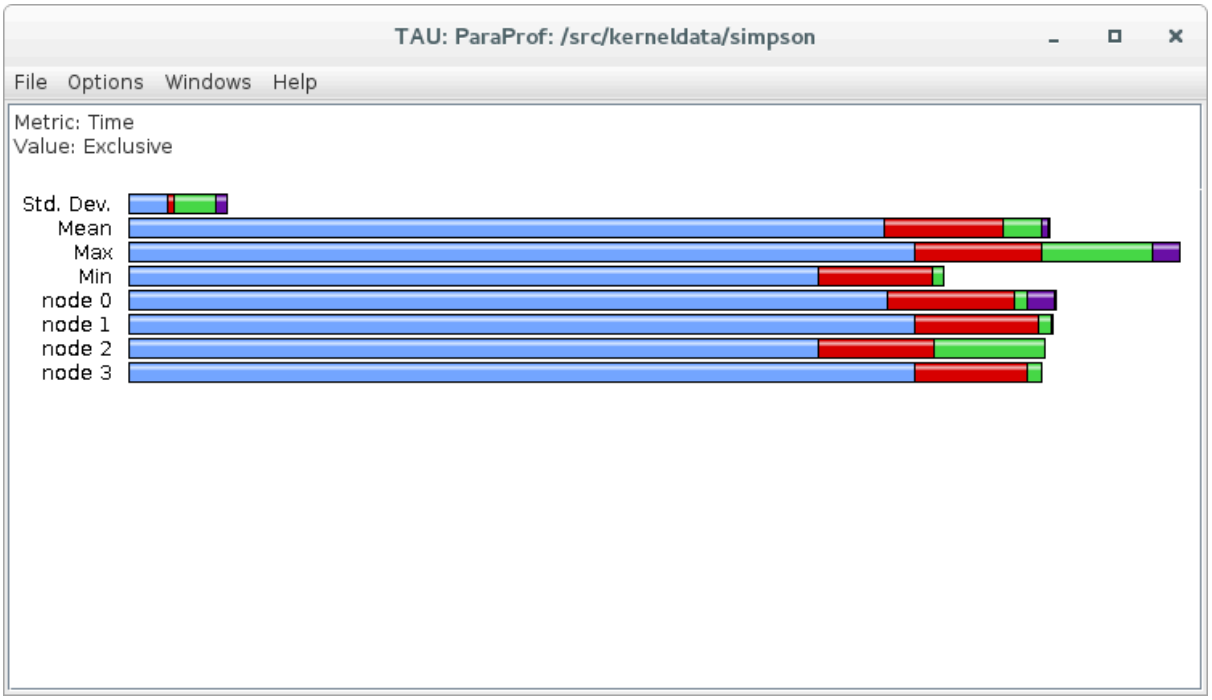


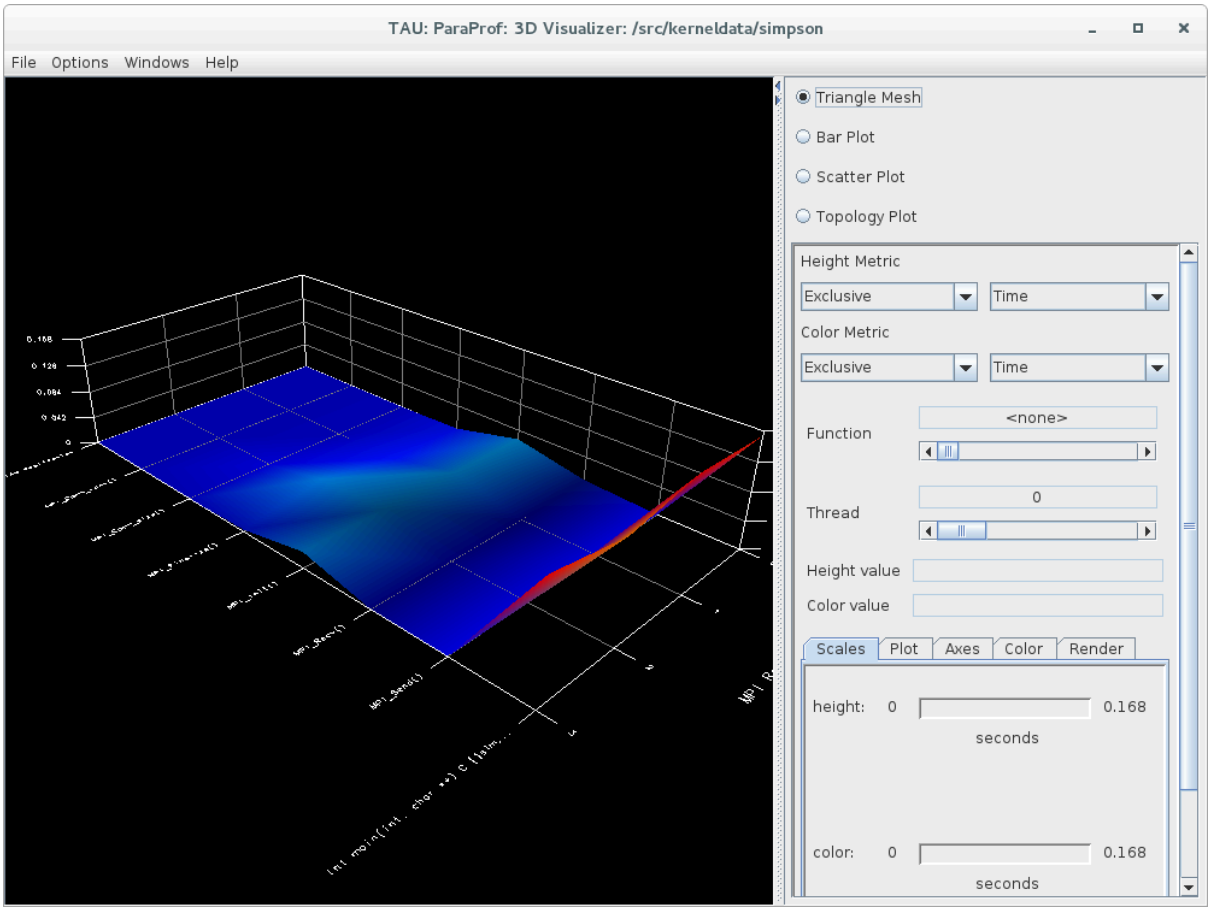
이 화면을 볼 수 있다.

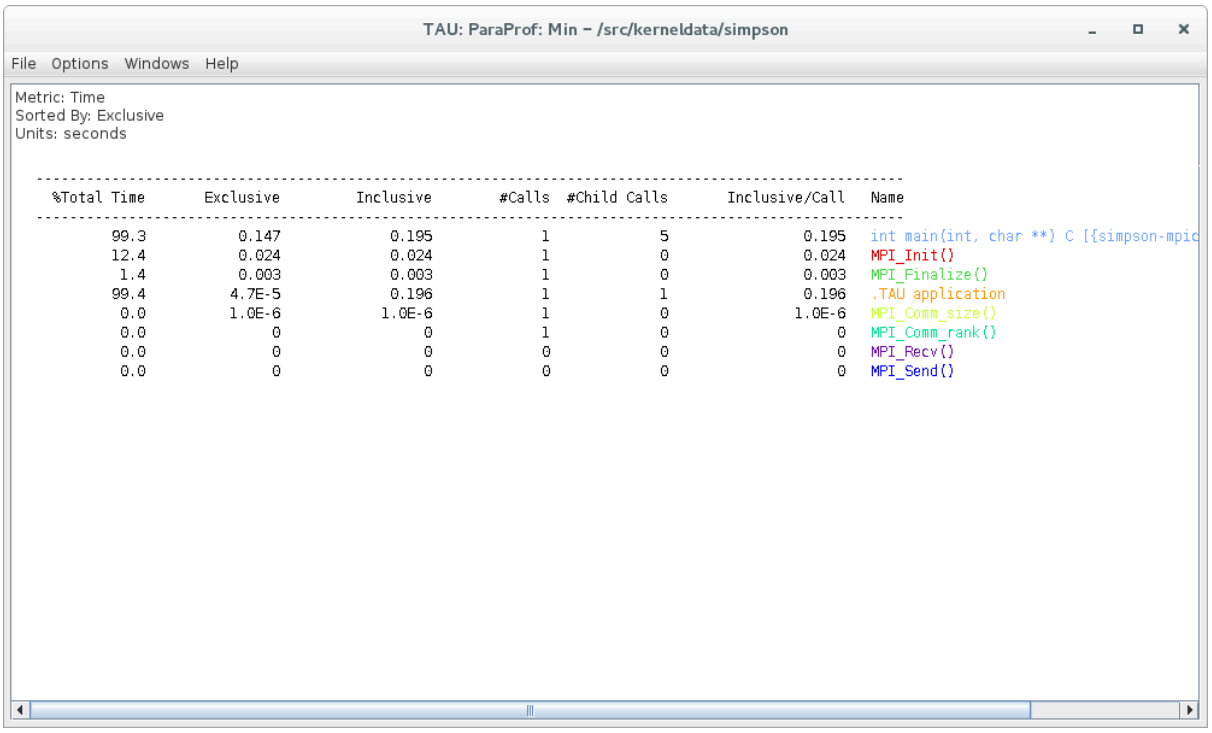
5.3.6. paraprof 사용

```
$ trace2profile tau.trc tau.edf
$ paraprof
```

이 명령어로 profile.0.0.0 ~ profile.3.0.0 파일을 얻을 수 있다. 그리고 paraprof를 실행한다.







Paraprof를 사용하여 좀 더 정밀하게 각 프로세스에서 함수별 실행되는 시간을 상세하게 볼 수 있다.

5.3.7. Example Test

tau를 설치하면 example 디렉토리가 생기는데 그 안의 test.py파일을 구동하여 테스트를 해 볼 수 있다.

```
$ cd example
$ test.py
```

거기서 어떤 부분이 설치되었고 어떤 예제가 동작하는지를 체크한 뒤에 이것으로 paraprof를 해보기 위한 테스트를 해 볼 수 있다.

5.3.8. Dockerfile

TAU를 자동 설치하는 Dockerfile은 아래와 같이 정리된다.

```
# TAU install
RUN mkdir /tau \
    && cd /tau \
    && wget https://www.cs.uoregon.edu/research/tau/tau_releases/tau-2.26.tar.gz \
    && wget https://www.cs.uoregon.edu/research/tau/pdt_releases/pdt-3.24.tar.gz \
    && tar -xzvf tau-2.26.tar.gz \
    && tar -xzvf pdt-3.24.tar.gz \
    && cd pdtoolkit-3.24 && ./configure && make && make install && cd .. \
    && cd tau-2.26 \
    && ./configure -c++=mpiCC  -cc=mpicc  -pdt=/tau/pdtoolkit-3.24  -mpi  -mpiinc=/usr/local/include \
        -mpilib=/usr/local/lib  -MPITRACE  -PROFILEPHASE  -PROFILEMEMORY  -PROFILE  -TRACE
-slog2 \
        -openmp  -CPUTIME  -LINUXTIMERS  -iowrapper \
    && make && make install && cd .. \
    && rm -rf tau-2.26.tar.gz pdt-3.24.tar.gz
ENV PATH /tau/tau-2.26/x86_64/bin:$PATH
ENV TAU_MAKEFILE
/tau/tau-2.26/x86_64/lib/Makefile.tau-memory-phase-mpi-pdt-openmp-profile-trace-mpitrace
```

5.4. Valgrind

Valgrind는 메모리 관리와 thread bugging, program profiling등에 사용되는 프로그램이다. 설치법과 간단히 메모리 누수를 막는 예제를 소개해 보고자 한다.

5.4.1. Installation

```
$ wget ftp://sourceware.org/pub/valgrind/valgrind-3.13.0.tar.bz2
$ tar xvjf valgrind-3.13.0.tar.bz2
$ cd valgrind-3.13.0
$ ./configure
$ make
$ make install
```

먼저 설치는 간단히 파일을 다운로드 하여 압축을 풀고 configure와 make 명령어로 설치한다.

5.4.2. Example

아래는 http://forum.falinux.com/zbxe/index.php?document_srl=528619 글에서 소개하는 메모리 누수를 막는 방법 예제를 빌려왔다.

sample_memory_leak.c

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>

void memory_leak( void)
{
   char *p_leak;
   int    ndx;

   p_leak  = malloc(10);

   for ( ndx = 0; ndx < 10; ndx++)
   {
      p_leak[ndx] = 'a';
   }
   for ( ndx = 0; ndx < 10; ndx++)
   {
      printf( "%c", p_leak[ndx]);
   }
}

int main( void)
{
   memory_leak();
}
```

이 프로그램은 malloc 이후 free 하지 않았기 때문에 memory leak 이 발생한다. 해당 코드를 컴파일 하여 아래와 같이 valgrind 로 실행한다.

```
[root@n6 vtest]# gcc -g test.c -o app_test
[root@n6 vtest]# valgrind --leak-check=yes ./app_test
==70== Memcheck, a memory error detector
==70== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==70== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==70== Command: ./app_test
==70==
aaaaaaaaaa==70==
==70== HEAP SUMMARY:
==70==     in use at exit: 10 bytes in 1 blocks
==70==   total heap usage: 1 allocs, 0 frees, 10 bytes allocated
==70==
==70== 10 bytes in 1 blocks are definitely lost in loss record 1 of 1
==70==    at 0x4C28C23: malloc (vg_replace_malloc.c:299)
==70==    by 0x40058E: memory_leak (test.c:10)
==70==    by 0x4005ED: main (test.c:24)
==70==
==70== LEAK SUMMARY:
==70==    definitely lost: 10 bytes in 1 blocks
==70==    indirectly lost: 0 bytes in 0 blocks
```

```
==70==    possibly lost: 0 bytes in 0 blocks
==70==  still reachable: 0 bytes in 0 blocks
==70==       suppressed: 0 bytes in 0 blocks
==70==
==70== For counts of detected and suppressed errors, rerun with: -v
==70== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

이처럼 얼마나 메모리가 lost 되었는지, 어느 line에서 leak가 되었는지를 확인 할 수 있다.

5.5. WireShark

WireShark는 네트워크 패킷, I/O를 살펴볼 수 있는 프로그램이다.

5.5.1. Installation

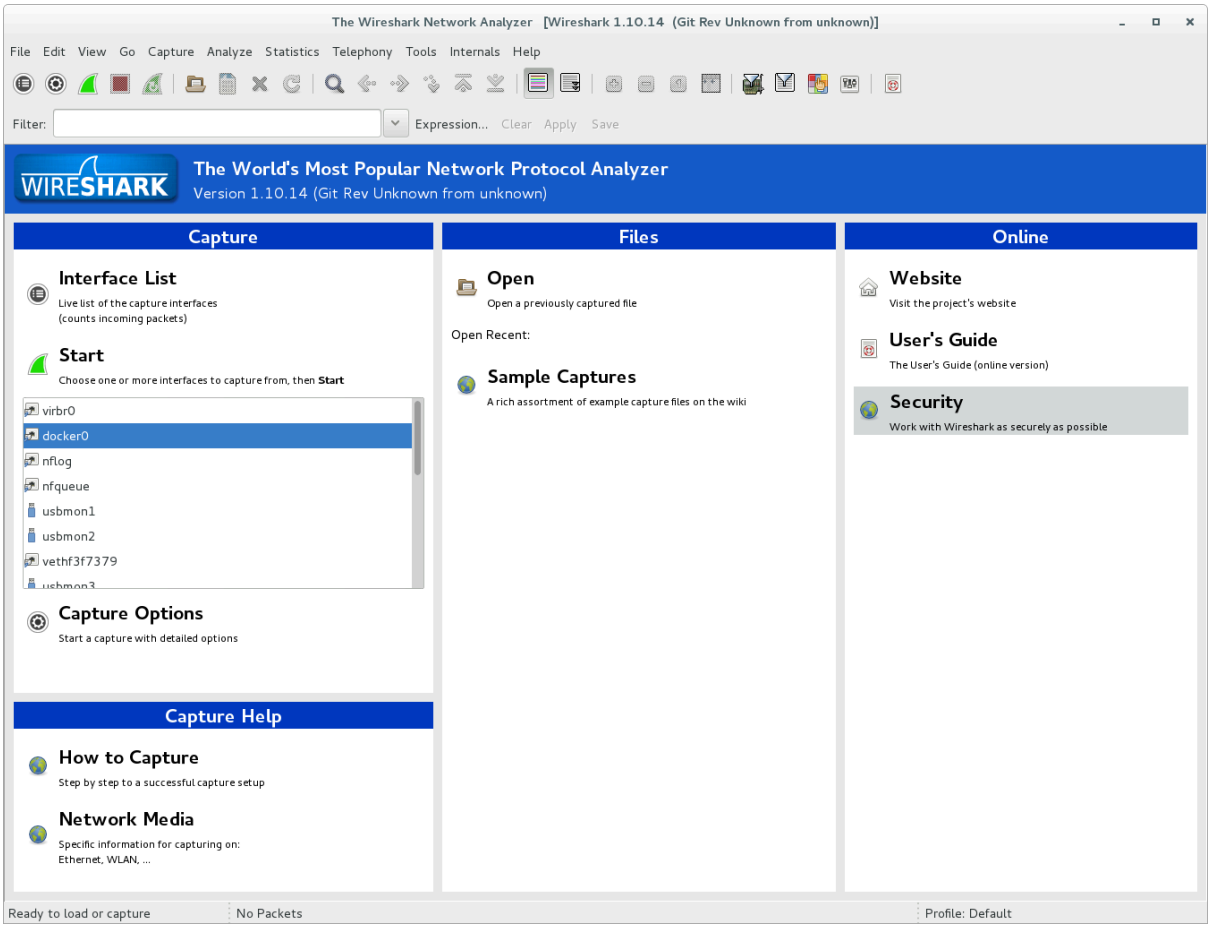
간단하게 yum 으로 설치 할 수 있다.

```
$ yum install wireshark
$ yum install wireshark-devel
$ yum install wireshark-gnome
```

5.5.2. wireshark를 실행

```
# wireshark
```

를 실행하면 (단 root 로 실행한다.)



(Start 버튼 아래 docker0이라는 것을 선택 하고 시작하면 된다.)

! 혹은 vethf3f7379와 같은 각 docker에서 돌아가는 듯한 interface도 하나하나 선택할 수 있다.

```
[root@rndc dockerdata]# brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.024204503fa8       no              veth6e3fa49
                                                        veth91a6e5a
                                                        vetha77bd99
                                                        vethca968e2
                                                        vethf4e0778
                                                        vethffaa58f
virbr0          8000.525400672138       yes             virbr0-nic
```

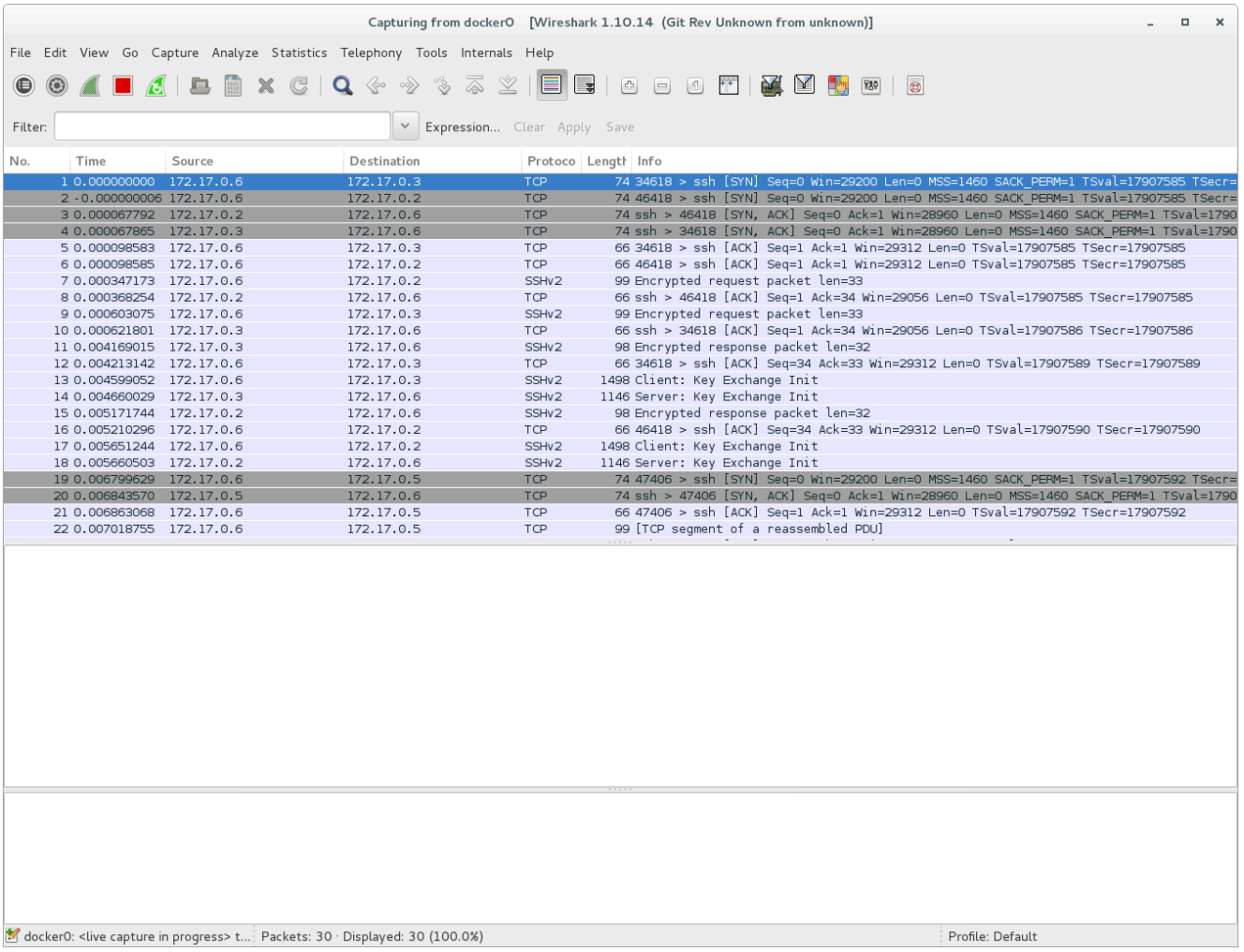
위와 같이 Bridge 상태를 확인해 볼 수 있다.

5.5.3. mpirun 실행해보기

먼저 아래의 명령어로 여러 대의 host를 사용하여 mpirun 을 실행하여 보자.

```
$ mpirun --allow-run-as-root -n 4 --hostfile mhostfile prog
```

현재 연산을 node는 172.17.0.6, MPI가 동작하는 node는 172.17.0.2 ~ 172.17.0.5 (이 중 172.17.0.2가 master고 나머지 셋은 slave이다)

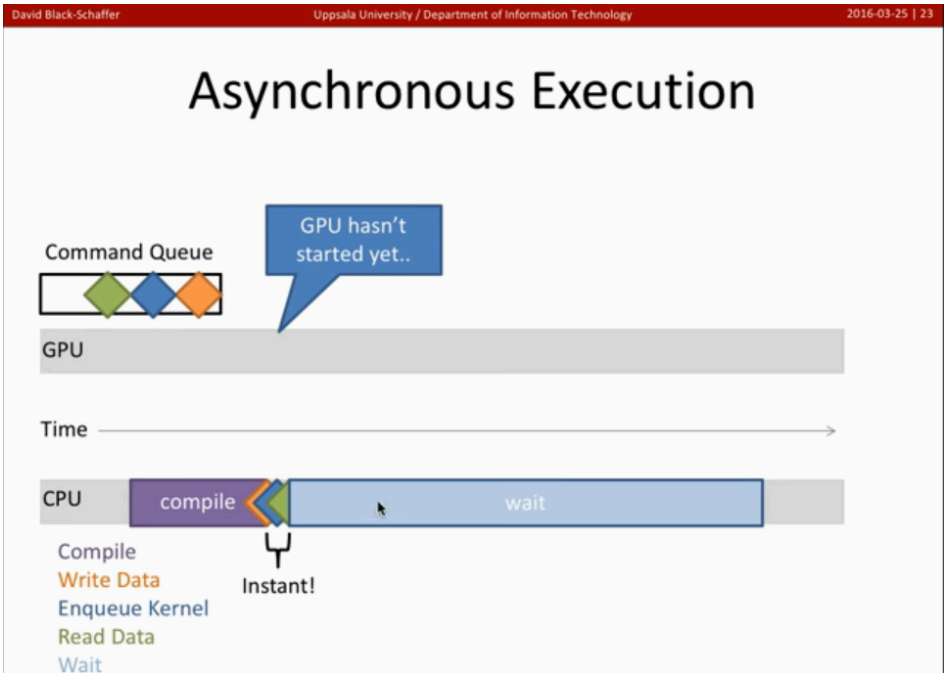


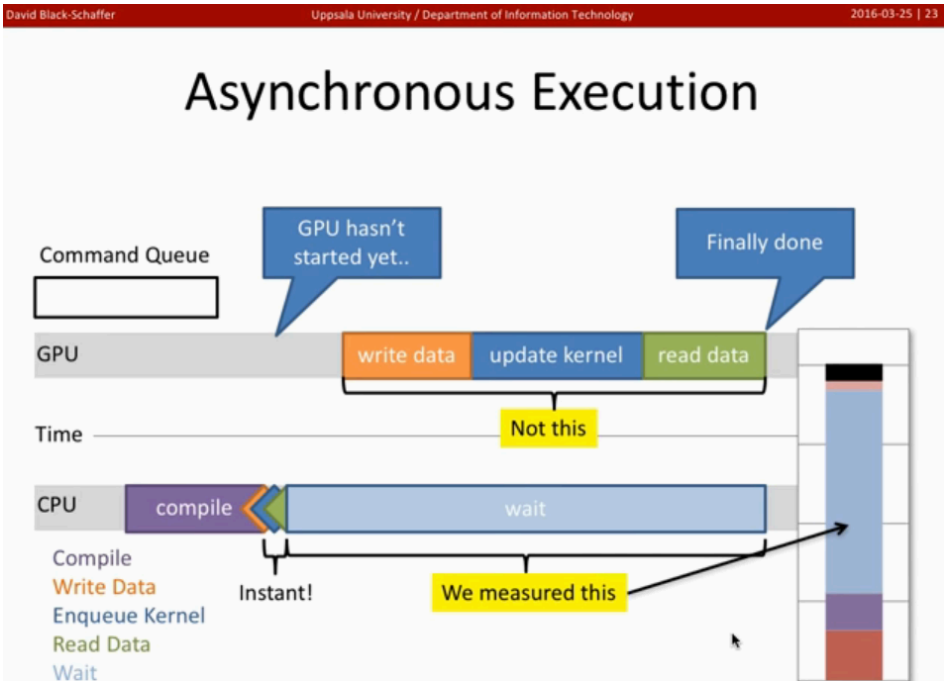
이와 같이 실행 된 결과를 볼 수 있다.

5.6. OpenCL Event Profiling

5.6.1. GPU에서 openCL 코드가 동작하는 방식과 기존의 Profiling 과정

아래의 슬라이드는 Uppsala Programming for Multicore Architectures Research Center의 Accelerating Code with OpenCL 강의자료의 일부를 발췌한 것이다. (출처: https://youtu.be/Ffn4ebXkViQ)

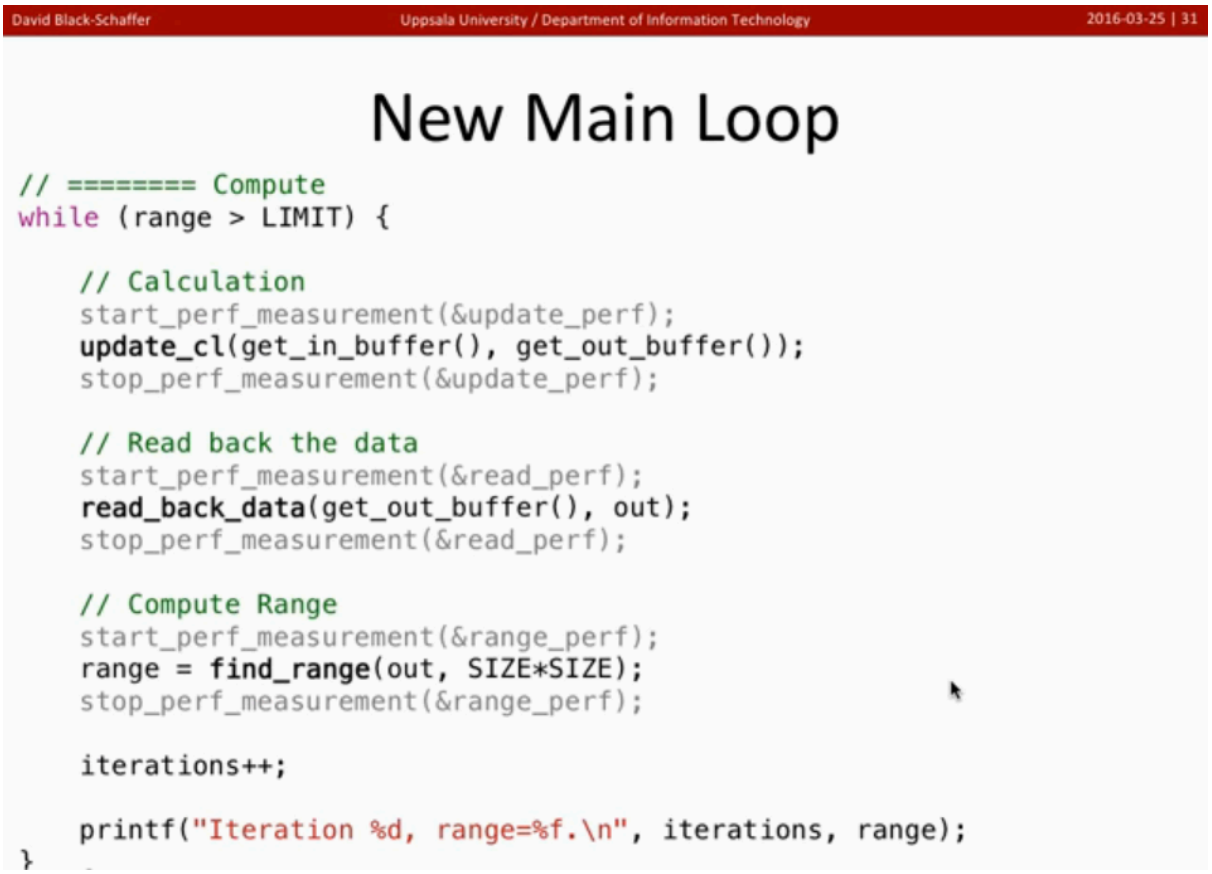




먼저 첫 번째 그림은 C main code 에서 .cl 파일을 컴파일 하고 Command Queue를 수행하는 것을 나타내고 있으며, 각각에 소요되는 시간을 나타내고 있다. 그림에서 볼 수 있듯이 컴파일이 가장 오랜 시간이 걸리고(clCreateProgramWithSource, clBuildProgram와 같은 함수들이 해당) Write Data, Enqueue Kernel, Read Data는 instant하게 일어나며, GPU Calculation Time에는 C main code가 waiting 상태가 되어 기본적으로는 프로파일링 되고 있지 않음을 보여주고 있다.

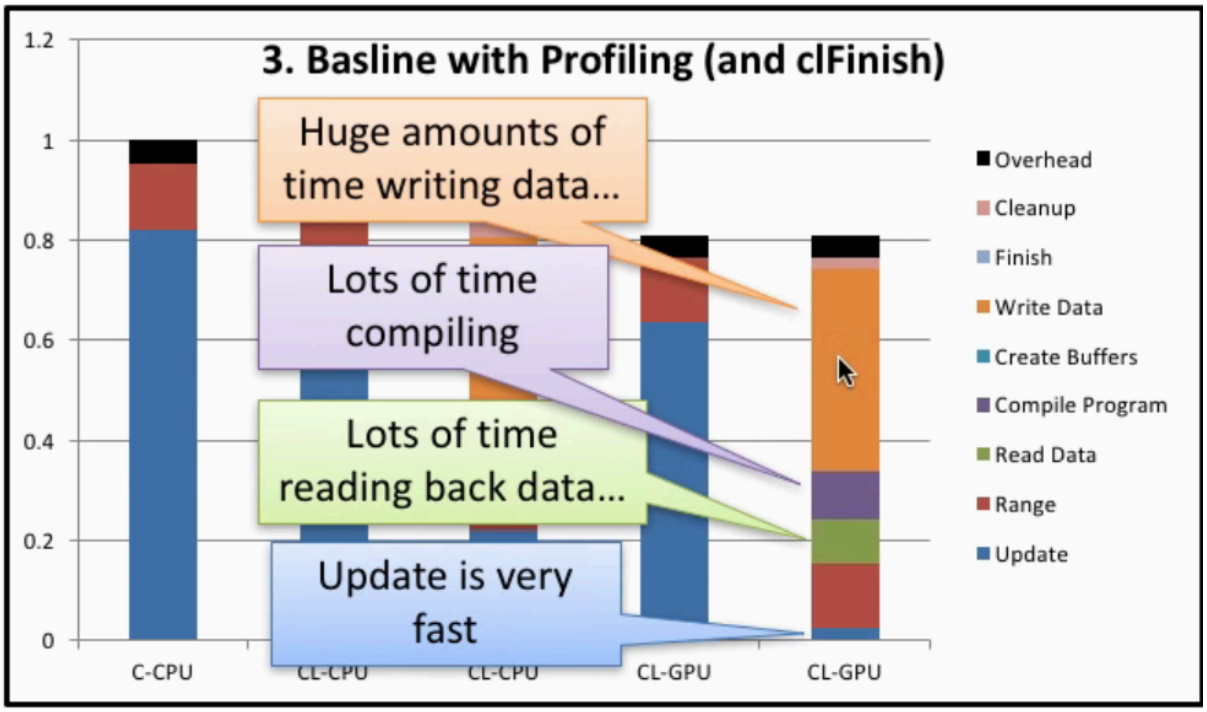
5.6.2. 각 과정마다 시간을 측정

기본적으로 GPU에서의 계산시간이 측정되지 않기 때문에 C의 main code에서 openCL의 function을 call할 때 wrapper를 만들어 시간을 측정하는 방식을 하여 시간을 측정을 해야 한다. 예를 들면 아래와 같이 각각의 함수마다 위 아래로 event에 대해 시간을 측정하는 것이다.



5.6.3. Profiling 결과

참고로 이 자료에서 말하는 Optimization 과정에서 얻게된 GPU Profiling 결과이며 대략적으로 GPU에서 어느 부분이 시간이 많이 소요되는지를 알 수 있다.



결과적으로 CPU에서 waiting time (GPU의 연산을 기다리는 과정)의 Detail 한 부분을 분석할 수 있었다. 그런데 정작 GPU 의 performance를 실질적으로 활용한 update function보다 read/write 부분의 시간이 더 많은 부분을 차지하였음을 알게 되었고 이를 개선하는 방법을 이후 소개하고 있다.

5.6.4. OpenCL Profiling 방법

실질적으로 OpenCL에서 시간을 측정할 수 있는 방법이다. 이는 C main 코드에서 각 event를 추적하여 그 시간을 측정 해 주어야 한다.

Profiling Example

```
cl_event prof_event;
cl_command_queue comm;

comm = clCreateCommandQueue(
    context, device_id,
    CL_QUEUE_PROFILING_ENABLE,
    &err);

err = clEnqueueNDRangeKernel(
    comm, kernel,
    nd, NULL, global, NULL,
    0, NULL, prof_event);

clFinish(comm);
err = clWaitForEvents(1, &prof_event );
```

```
cl_ulong start_time, end_time;
size_t return_bytes;

err = clGetEventProfilingInfo(
   prof_event,
    CL_PROFILING_COMMAND_QUEUED,
    sizeof(cl_ulong),
    &start_time,
    &return_bytes);

err = clGetEventProfilingInfo(
    prof_event,
    CL_PROFILING_COMMAND_END,
    sizeof(cl_ulong),
    &end_time,
    &return_bytes);

run_time =(double)(end_time - start_time);
```

1. 먼저 Profile Enabled option을 주어 Command Queue를 발생시켜야 한다.
   → `command_queue = clCreateCommandQueue(context, devices[deviceUsed], CL_QUEUE_PROFILING_ENABLE, &err);`
2. 또한 확실하게 task가 끝난 상태로 진행할 수 있도록 clFinish를 한다. (위 그림에서는 표시가 안되어있음)
   → `clFinish(command_queue);`
3. 이후 해당 event가 연결된 kernel을 실행한다.
   → `err = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, workGroupSize, NULL, 0, NULL, &event);`
4. kernel이 확실히 완료되도록 기다린다.
   → clWaitForEvents(1 , &event);
5. 마지막으로 Profiling 된 데이터를 가져온다.

```
→   cl_event event;
    cl_ulong time_start, time_end;
    double total_time;
    clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
    sizeof(time_start), &time_start, NULL);
    clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(time_end),
    &time_end, NULL);
    total_time = time_end - time_start;
    printf("\nExecution time in milliseconds = %0.3f ms\n", (total_time /
    1000000.0) );
```

5.6.5. Example Source Code

OpenCL Profiling 이를 적용한 Simpson's Rule 계산 예제 코드: (*simpson-mpicl.cl 코드는 동일함)

simpson-mpicl.c

```
1  // This program tests Simpson theorem using OpenCL
2
3  // MPI Includes
4  #include <mpi.h>
5
6  // System includes
7  #include <stdio.h>
8  #include <stdlib.h>
9
10 // OpenCL includes
11 #include <CL/cl.h>
12
13 #define MAX_SOURCE_SIZE (0x100000)
14 // Define min macro
15 #define min(X, Y) ((X) < (Y) ? (X) : (Y))
16
17 void equal_load(int, int, int, int, int*, int*);
18 int main(int argc, char** argv)
19 {
20     int myid, nproc;
21     int n = 500000, i;
22     float sum = 0.L;
23     float aa = 0.L;
24     float bb = 1.L;
25     float time_start, time_end;
26     float h = (bb-aa)/(float)n;
27     int istart, ifinish;
28
29     // Initialize the MPI environment
30     MPI_Init(NULL, NULL);
31
32     // Get the number of processes
33     MPI_Comm_size(MPI_COMM_WORLD, &nproc);
34
35     // Get the rank of the process
36     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
37
38     // Check starting time
39     time_start = MPI_Wtime();
40
41
42     // Load the source code containing the kernel
43     FILE *fp;
44     char fileName[] = "./simpson-mpicl.cl";
45     char *source_str;
46     size_t source_size;
47
48     /* Load the source code containing the kernel*/
49     fp = fopen(fileName, "r");
50     if (!fp) {
51         fprintf(stderr, "Failed to load kernel.\n");
52         exit(1);
53     }
54     source_str = (char*)malloc(MAX_SOURCE_SIZE);
```

```
55    source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
56    fclose(fp);
57
58    // Load equally
59    equal_load(1, n-1, nproc, myid, &istart, &ifinish);
60    printf("(%d / %d) calculate from %d to %d\n", myid+1, nproc, istart, ifinish);
61
62    // num elements
63    int nelem = ifinish - istart + 1;
64
65    // This code executes on the OpenCL host
66    // Host data⊏
67    float *X=NULL; // Input array
68    float *Y=NULL; // Output array
69
70    // Compute the size of the data
71    size_t datasize=sizeof(float)*nelem;
72
73    // Allocate space for input/output data
74    X=(float*)malloc(datasize);
75    Y=(float*)malloc(datasize);
76
77    // Initialize the input data
78    for(i=0; i<nelem; i++){
79       X[i]= aa + h*(float)(i+istart);
80    }
81
82    // Use this to check the output of each API call
83    cl_int status;
84
85    printf("A%d\n", myid);
86    // Retrieve the number of platforms
87    cl_uint numPlatforms=0;
88    status=clGetPlatformIDs(0, NULL, &numPlatforms);
89
90    printf("B numPlatforms = %d %d\n", myid, numPlatforms);
91    // Allocate enough space for each platform
92    cl_platform_id *platforms=NULL;
93    platforms=(cl_platform_id*)malloc(
94       numPlatforms*sizeof(cl_platform_id));
95
96    printf("C%d\n", myid);
97    // Fill in the platforms
98    status=clGetPlatformIDs(numPlatforms, platforms, NULL);
99
100   printf("D%d\n", myid);
101   // Retrieve the number of devices
102   cl_uint numDevices=0;
103   status=clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, 0,
104      NULL, &numDevices);
105
106   printf("E%d\n", myid);
107   // Allocate enough space for each device
108   cl_device_id *devices;
109   devices=(cl_device_id*)malloc(
110      numDevices*sizeof(cl_device_id));
111
112   printf("F%d\n", myid);
113   // Fill in the devices
114   status=clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL,
115      numDevices, devices, NULL);
116
```

```
117     printf("G%d\n", myid);
118     // Create a context and associate it with the devices
119     cl_context context;
120     context=clCreateContext(NULL, numDevices, devices, NULL,
121        NULL, &status);
122
123     printf("H%d\n", myid);
124     // Create a command queue and associate it with the device
125     cl_command_queue cmdQueue;
126     cmdQueue=clCreateCommandQueue(context, devices[0], CL_QUEUE_PROFILING_ENABLE,
127        &status);
128
129     // Create a buffer object that will contain the data
130     // from the host array A
131     cl_mem bufX;
132     bufX=clCreateBuffer(context, CL_MEM_READ_ONLY, datasize,
133        NULL, &status);
134
135     // Create a buffer object that will contain the output data
136     // from the host array Y
137     cl_mem bufY;
138     bufY=clCreateBuffer(context, CL_MEM_WRITE_ONLY, datasize,
139        NULL, &status);
140
141     // Write input array A to the device buffer bufferA
142     status=clEnqueueWriteBuffer(cmdQueue, bufX, CL_FALSE,
143        0, datasize, X, 0, NULL, NULL);
144
145     // Write input array A to the device buffer bufferB
146     status=clEnqueueWriteBuffer(cmdQueue, bufY, CL_FALSE,
147        0, datasize, Y, 0, NULL, NULL);
148
149     // Create a program with source code
150     cl_program program=clCreateProgramWithSource(context, 1,
151        (const char**)&source_str, NULL, &status);
152
153     // Build (compile) the program for the device
154     status=clBuildProgram(program, numDevices, devices,
155        NULL, NULL, NULL);
156
157     // Create the vector addition kernel
158     cl_kernel kernel;
159     kernel=clCreateKernel(program, "func", &status);
160
161     // Associate the input and output buffers with the kernel
162     status=clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufX);
163     status=clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufY);
164
165     // Define an index space (global work size) of work
166     // items for execution. A workgroup size (local work size)
167     // is not required, but can be used.
168     size_t globalWorkSize[1];
169
170     // There are 'nelem' work-items
171     globalWorkSize[0]=nelem;
172
173     cl_event event;
174     // Execute the kernel for execution
175     status=clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL,
176        globalWorkSize, NULL, 0, NULL, &event);
177
178     // Read the device output buffer to the host output array
```

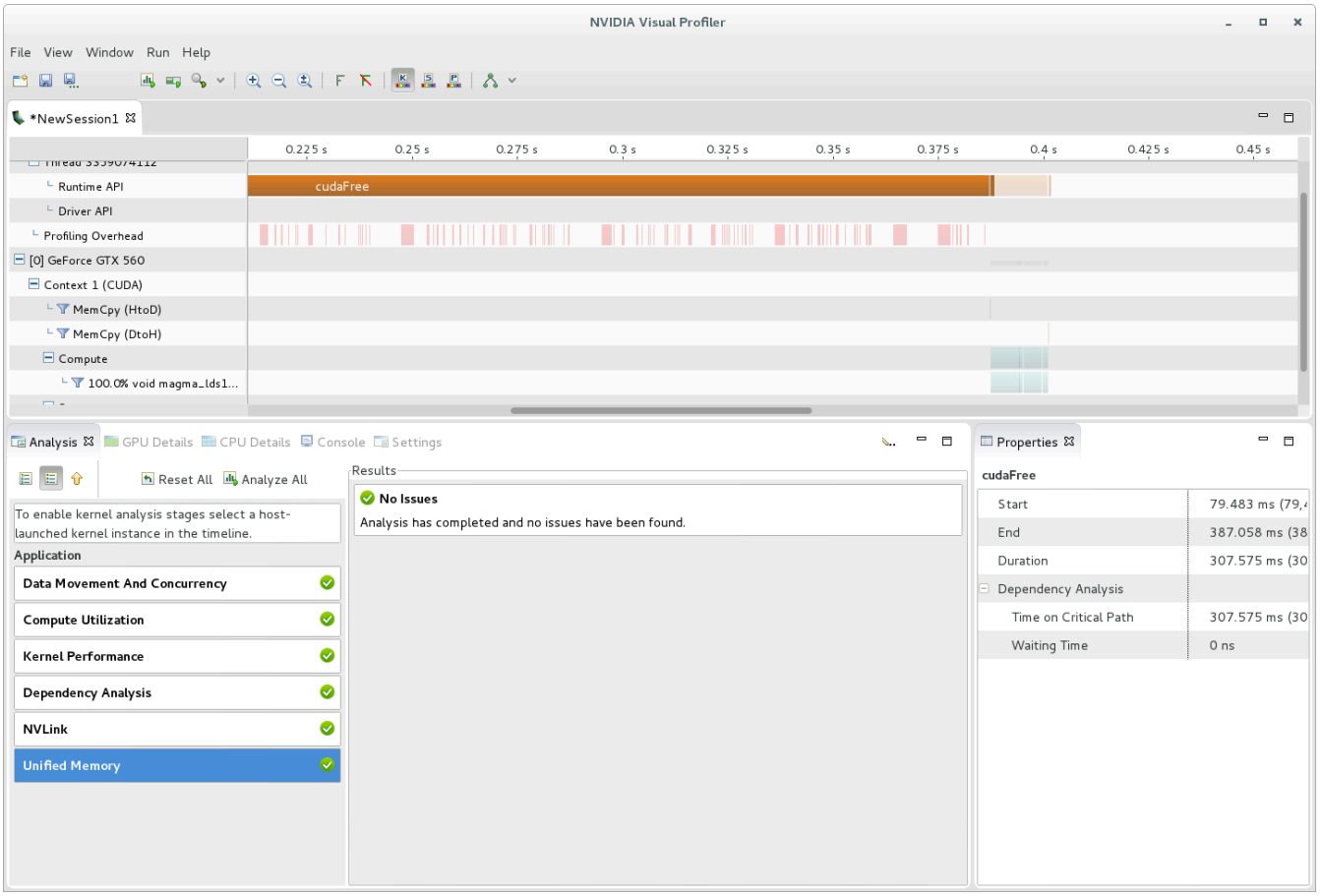
```
179     clEnqueueReadBuffer(cmdQueue, bufY, CL_TRUE, 0,
180         datasize, Y, 0, NULL, NULL);
181     // Reduce the output
182     for(i=0; i<nelem; i++) {
183         if ((myid == 0 && i == 0) || (myid == nproc-1 && i == nelem-1)) sum += Y[i];
184         else if ((istart + i)%2 == 0) sum += Y[i] * 2;
185         else sum += Y[i] * 4;
186     }
187
188     sum = sum * h / 3.L;
189
190     if(nproc > 1) {
191         if(myid > 0) { // Slave
192             MPI_Send(&sum, 1, MPI_FLOAT, 0, 42, MPI_COMM_WORLD);
193         }
194         else { // Master
195             float total = sum;
196             float tmp;
197             for(i=1; i<nproc; i++){
198                 MPI_Recv(&tmp, 1, MPI_FLOAT, i, 42, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
199                 total += tmp;
200             }
201             printf("Result = %f\n", total);
202         }
203     }
204     else {
205         printf("Result = %f\n", sum);
206     }
207
208     time_end = MPI_Wtime();
209     if(myid==0) {
210         float timer = time_end - time_start;
211         printf("%14.5g s\n", timer);
212     }
213
214     cl_ulong t_start, t_end;
215     t_start = t_end = 0;
216     double t_time;
217
218     printf("%llu, %llu\n", t_start, t_end);
219     clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &t_start, NULL);
220     clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &t_end, NULL);
221     t_time = t_end - t_start;
222     printf("%llu, %llu\n", t_start, t_end);
223     printf("\nExecution time in milliseconds = %0.3f ms\n", (t_time / 1000000.0) );
224
225     // Free OpenCL resources
226     clReleaseKernel(kernel);
227     clReleaseProgram(program);
228     clReleaseCommandQueue(cmdQueue);
229     clReleaseMemObject(bufX);
230     clReleaseMemObject(bufY);
231     clReleaseContext(context);
232
233     // Free host resources
234     free(source_str);
235     free(platforms);
236     free(devices);
237     free(X);
238     free(Y);
239
240     // Finalize the MPI environment.
```

```
241     MPI_Finalize();
242
243     return 0;
244 }
245
246 void equal_load(int n1, int n2, int nproc, int myid, int *istart, int *ifinish) {
247     int iw1, iw2;
248     iw1 = (n2-n1+1)/nproc;
249     iw2 = (n2-n1+1)%nproc;
250     *istart = myid*iw1 + n1 + min(myid, iw2);
251     *ifinish = *istart + iw1 - 1;
252     if (iw2 > myid) *ifinish = *ifinish + 1;
253     if (n2 < *istart) *ifinish = *istart - 1;
254 }
```

5.7. NVIDIA Visual Profiler (NVVP)

5.7.1. Example

이번 프로젝트에서는 실질적으로 CUDA를 사용하지 않기 때문에 NVVP를 사용할 일은 없었다. 다만, 기존에는 프로그램을 해킹하여 OpenCL을 NVVP가 프로파일링 하는 방법이 있었다고 한다만 8버전 패치가 되며 CUDA만 지원하게 되었다. 아래는 CUDA Example을 실행시켰을 때 GPU 내에서 각 event 별로 얼마나 시간이 걸리는 지를 시각적으로 보여주는 스크린샷이다.



6. Conclusion

프로토타이핑의 단계이기는 하지만 만들어진 Docker이미지를 기반으로 실제 노드들에 연결시켜 -p로 포트바인딩을 한 이후 Slurm으로 연결하여 작업을 전달할 수 있었다. 또한 cgroup에서도 cpuset을 시켜 특정 코어만 사용하게 해 본다거나 Profiling Tool들을 조사하고 설치하여 사용해 봄으로써 앞으로의 Scheduler를 디자인에 대해 고찰해 볼 수 있었다. 다만 NVIDIA-docker 사용에 관하여 직접 git을 통해 메일을 남기거나 Slurm이나 Cgroup에 대한 자료가 많이 부족하여 실제로 관계된 slurm-devel의 google group에 문의하여 Trouble Shooting을 진행하여야 하였는데, 그 과정이 벙거로웠지만 한편으로는 HPC분야의 전문가가 드물어 블루오션일 수 있다는 생각이 들었다.

이번 프로젝트에서 힘들었던 점은, Slurm 설치와 Cgroup등 HPC 관련된 자료가 미흡하였고, 있다고 하더라도 추상적이거나 높은 수준의 지식을 요구했다. 특히 Cgroup개념에 대해서는 Kernel의 기본 구조나 운영체제적인 기초 지식까지 요구되어 어려웠다. 또한 CentOS 버전이 아닌 다른 Linux 기반 OS(Ubuntu나 Debian, RHEL 등)에서 작동한다고 소개하는 문서를 CentOS에 맞게 재가공 하는 과정에 있어서도 Trouble Shooting이 많았다. 그리고 문제를 해결하는 과정에 있어서 본질적으로 문제를 해결해야 하는 목적을 계속 되내어야 했다. 자칫 문제의 틀에서 벗어나 사소한 부분을 조사하느라 시간을 낭비하거나 혹은 중요한 문구를 지나쳐 어렴풋이 추측으로 문제를 해결하려 하기도 하였는데 그 부분을 조심해야 할 것 같다.

이번 인턴내용을 통해 자동 프로파일링 툴로 리소스 사용량을 분석하여 특정 Spec의 컴퓨터에서의 계산 시간 예상과 이를 활용한 고급 Scheduler 제작하거나 Ganglia나 Cgroup과 같은 실시간 모니터링 기술을 Scheduler에 연동하여 임의의 job에 대해 더 빠르게 계산할 수 있는 알고리즘에 대한 연구, 모바일 환경에서의 리소스 모니터링과 Scheduler 제작등에 대해 고려해 볼 수 있을 것이라 생각한다. 컴퓨터 OS나 Architecture와 같은 기초적인 분야와 맞닿아 있는 만큼 경쟁력을 갖는다면 다양한 분야에도 쓰일 수 있는 기술이라고 생각한다.

ㅇ 소감 및 건의사항(분량제한 없음)

이번 인턴 경험을 통해 HPC분야에 대해 접해보고 실제 슈퍼컴퓨터들에서 여러개의 노드들을 어떻게 연결하고 효율적으로 그 자원들을 관리하는지에 대해 실험해 볼 수 있는 좋은 기회가 되었다. 특히 Cgroup과 관련하여 커널 단계에서 시스템 자원을 단순히 파일시스템처럼 제한을 줄 수 있다는 것도 혁신적인 기술이라고 생각한다. 이것을 기반으로 LXC(Linux Container)라는 개념이 나왔고 거기서 더 발전하여 $95M 기업 가치를 가지는 Docker라는 신기술이 발전될 수 있었는데, Cgroup에 대해 자세히 공부한다면 앞으로 내가 하고자하는 HPC분야나 OS분야에 큰 도움이 될 것이라고 기대된다.

그리고 이번 인턴 경험을 통해 국내 전문가가 부족함을 크게 느꼈다. 예전에 슈퍼컴퓨터 병렬처리 프로그래밍 대회에 출전한 경험이 있는데, 그 때 서울대의 천둥 슈퍼컴퓨터를 사용했었는데 국산 기술로 그것을 만들었다는 것이 대단하고 존경스럽다고 생각들었다. 비록 아주 상용화된 소프트웨어를 제작하지는 못하였지만 OpenCL과 CUDA의 Translator를 Open Source로 배포하는 것도 좋은 시도라고 생각하고 꾸준히 도전한다는 인상을 받았다. 그러나 국내에는 아직 제대로 된 전문가가 드물다고 한다. KISTI나 ETRI와 같은 연구기관에서도 해외의 기술을 구입해서 사용하는 정도로만 진행되고 있으며 실질적으로 프로그래밍을 할 수 있는 사람은 드물다고 한다.

이번 보고서의 제목인 “Platform”은 “Framework”와는 구별되어 소프트웨어의 뼈대 구조를 넘어 그 실행환경까지 마련하는 것을 의미한다. 단순히 추상화 단계에 지나지 않는 것이 아니라 Slurm을 실제로 설치하고 여러가지 Profiling Tool들을 설치함으로써 실제로 차후에 Scheduler를 설계하기 위한 기반을 고려하며 인턴십 프로그램을 진행하였다. 이번 경험은 앞으로 내가 연구해 나가는 분야에 있어서도 여러모로 자산이 될 것이라고 생각한다. 앞으로도 (주) 쎄트렉아이가 더 발전하여 한국의 우주기술을 굳건히 지키고 직원과 임원진이 행복하게 상생할 수 있는 더 발전된 회사로 나아갔으면 좋겠다. 완벽한 인턴경험이라고 말할순 없지만 나름대로 만족스러웠고 단순히 아르바이트 경험이 아닌, 내가 스스로 직접 답을 찾아 나가고 연구해 가며 고민해 나갈 수 있던 좋은 경험이었다고 생각한다.