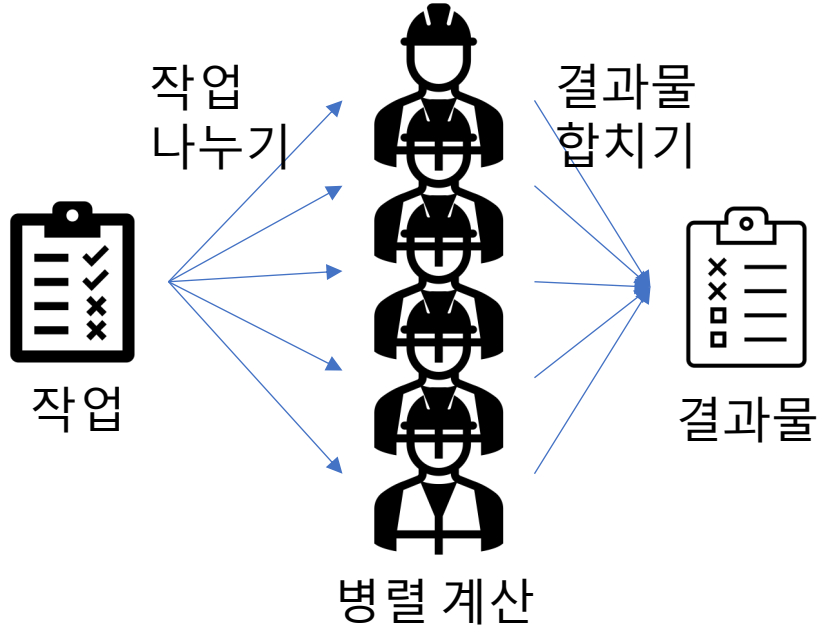
# 병렬 처리를 활용한 대용량 계산 방법

정인석
(주)스탠다임

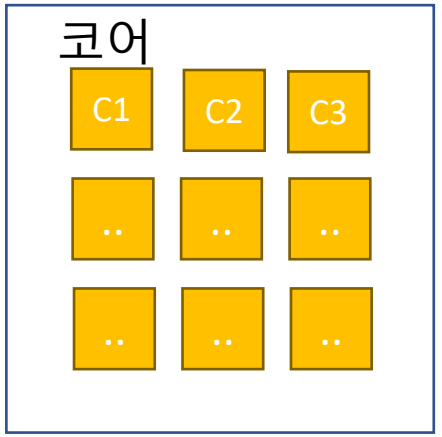
# 병렬 처리의 목적 및 방향

- **문제점: 계산 처리에 시간이 오래 걸린다.**
- 단일 컴퓨터의 성능이 올라가면 (Hz 상승, 1 GHz vs 2 GHz) 계산 속도가 빨라진다. 이 경우 기존의 프로그램을 별도로 수정할 필요가 없이 계산 시간을 줄일 수 있다.
- 무어(Moore) 의 법칙의 한계 → 반도체 집적도의 물리적 제약으로 인해 단일 코어 CPU의 성능에는 한계치가 존재한다.
- 멀티 코어 및 멀티 노드의 컴퓨팅 자원을 활용의 필요성이 발생하게 된다.
- **멀티 코어 및 멀티 노드에서 작업을 나누어서 진행할 수 있도록 프로그래밍 하는 과정이 필요**

# 병렬 처리 방식

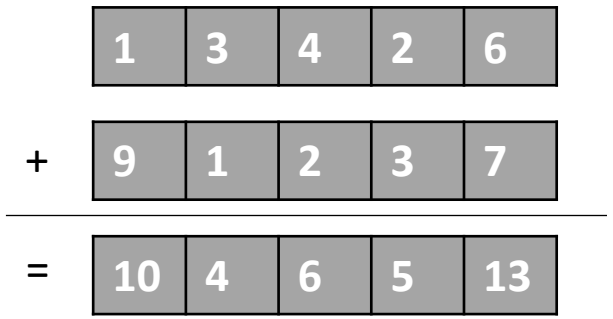
- CPU 병렬 처리
  - 멀티 프로세싱 (Multiprocessing)
    - : 프로세스 아이디가 각각 부여되는 복수의 작업들을 활용하는 병렬 처리
    - : 각 프로세스들 간에는 메모리를 공유하지 않음
  - 멀티 스레딩 (Multithreading)
    - : 단일 프로세스 내에서 멀티 코어를 활용하는 병렬 처리 방식
    - : 단일 프로세스이므로 작업들 간에 메모리를 공유함

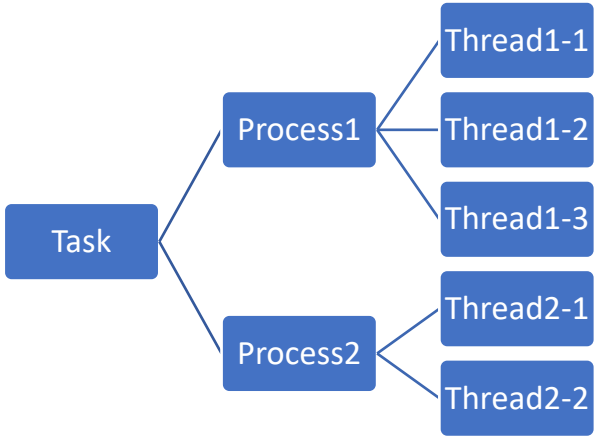


CPU 중앙 처리 장치

| 시간 | 작업 | 할당된 코어 |
|---|---|---|
| $t_0$ | P1 – task 1 | Core 1 |
| $t_1$ | P2 – task 1 | Core 2 |
| $t_2$ | P3 – task 1 | Core 1 |
| $t_3$ | P1 – task 2 | Core 1 |
| $t_4$ | P1 – task 3 | Core 3 |
| … | … | … |

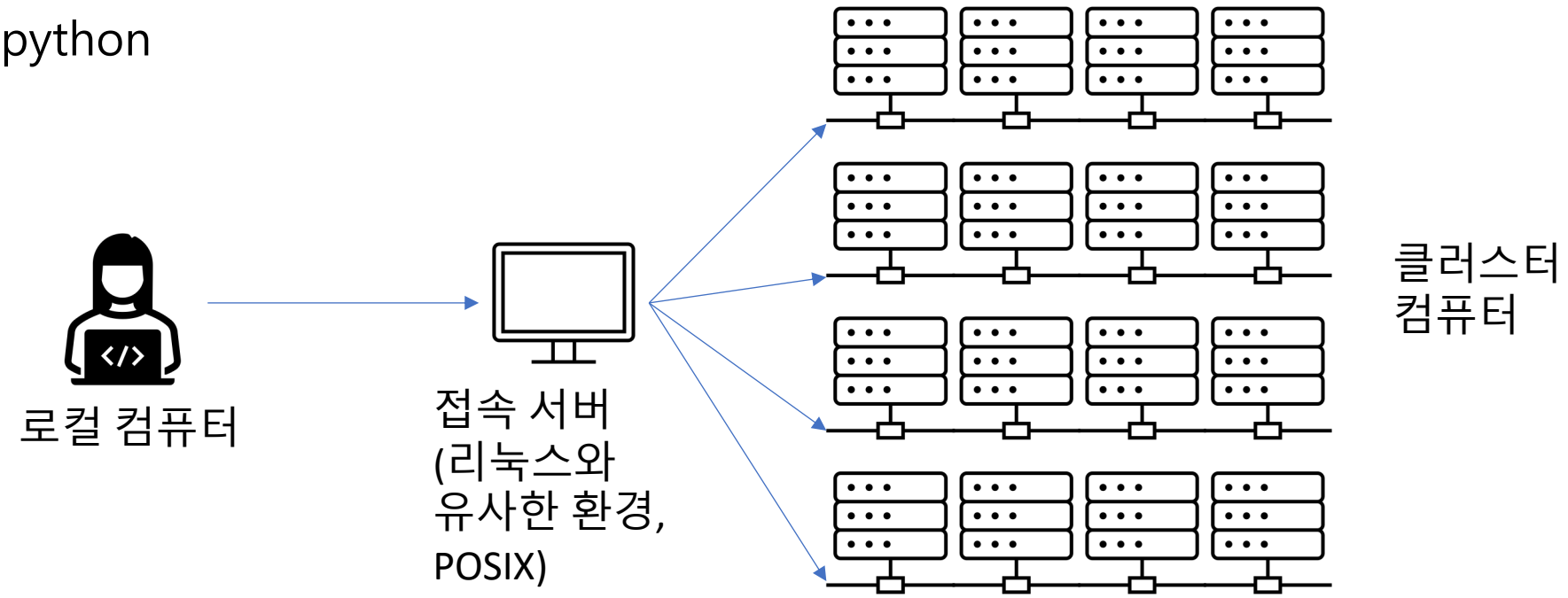
- GPU 병렬 처리
  - 벡터(텐서) 단위의 연산
  - 병렬 처리 방식이 제한적
  - TensorFlow, pyTorch



# 작업 환경 소개

- 간단한 계산은 로컬 컴퓨터에서 실행 가능하나 대량의 계산은 클러스터 컴퓨터 (슈퍼 컴퓨터)를 활용하는 경우가 대부분이다
- 리눅스와 유사한 터미널 환경 (리눅스 및 Mac OS)
- 윈도우 사용시 WSL (Window Subsystem for Linux) 설치
  - WSL 설치 방법 (https://docs.microsoft.com/ko-kr/windows/wsl/install)
- Terminal: bash 또는 zsh
- Anaconda python



# Tanimoto Similarity 계산 예제

- 비 병렬 작업 예제 (폴더: tanimoto)
  1. ./calc_fingerprint.py
  2. ./calc_tanimoto.py
- 화학적 유사도 (https://en.wikipedia.org/wiki/Chemical_similarity)
- Tanimoto 유사도 (Jaccard index)

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}.$$

| | | | | | | |
|---|---|---|---|---|---|---|
| Fingerprint_A | 0 | **1** | **1** | 0 | 0 | **1** |
| Fingerprint_B | 0 | 0 | **1** | **1** | **1** | **1** |

$$|A| = 3 \quad |B| = 4 \quad |A \cap B| = 2$$

# 단순 병렬화

- 단순 병렬 작업 = 작업 끼리의 상호 관계가 없는 경우
- Embarassingly parallel (https://en.wikipedia.org/wiki/Embarrassingly_parallel)
- 작업 예제 (폴더: tanimoto_split)

- 방법

1. 작업을 나누고 각 작업자에 업무를 할당
2. 각 작업을 수행
3. 작업의 결과를 합침

- 학습
  - Shell 에서 백그라운드 프로세스를 생성하기 위해서는 끝에 & 를 붙여준다
  - 백그라운드 작업이 종료되기를 기다리는 명령어 'wait'
  - 작업을 나누고 합치는 시간에 대한 고려

# Job 나누기

```
import numpy
split_data = numpy.array_split(data, size)
```

```
>> data = [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
>> split_data = numpy.array(data, 3)
>> print(split_data)
```

```
[array([1, 2, 3]), array([4, 5, 6]), array([7, 8, 9])]
```

**% (나머지) 연산자**

```
import numpy as np

n_split = 4
split_data = [ [] for x in range(n_split) ]

jobs = np.random.random(100)

for i, job in enumerate(jobs):
    ind = i % n_split
    split_data[ind].append(job)
```

# Python Multiprocessing 모듈을 활용한 병렬화

- 파이썬에서 프로세스를 생성하기 위한 표준 기법
- multiprocessing module vs multithreading module: multithreading 은 메모리를 공유할 수 있다는 장점이 있으나 python global interpreter lock (GIL) 으로 인하여 계산 시간에 대한 이득이 없음

```
import multiprocessing

p = multiprocessing.Process()
m = multiprocessing.Manager()
with multiprocessing.Pool(nproc) as p:
    p.map(func, arg)
```

← 프로세스 생성

← 매니저 생성

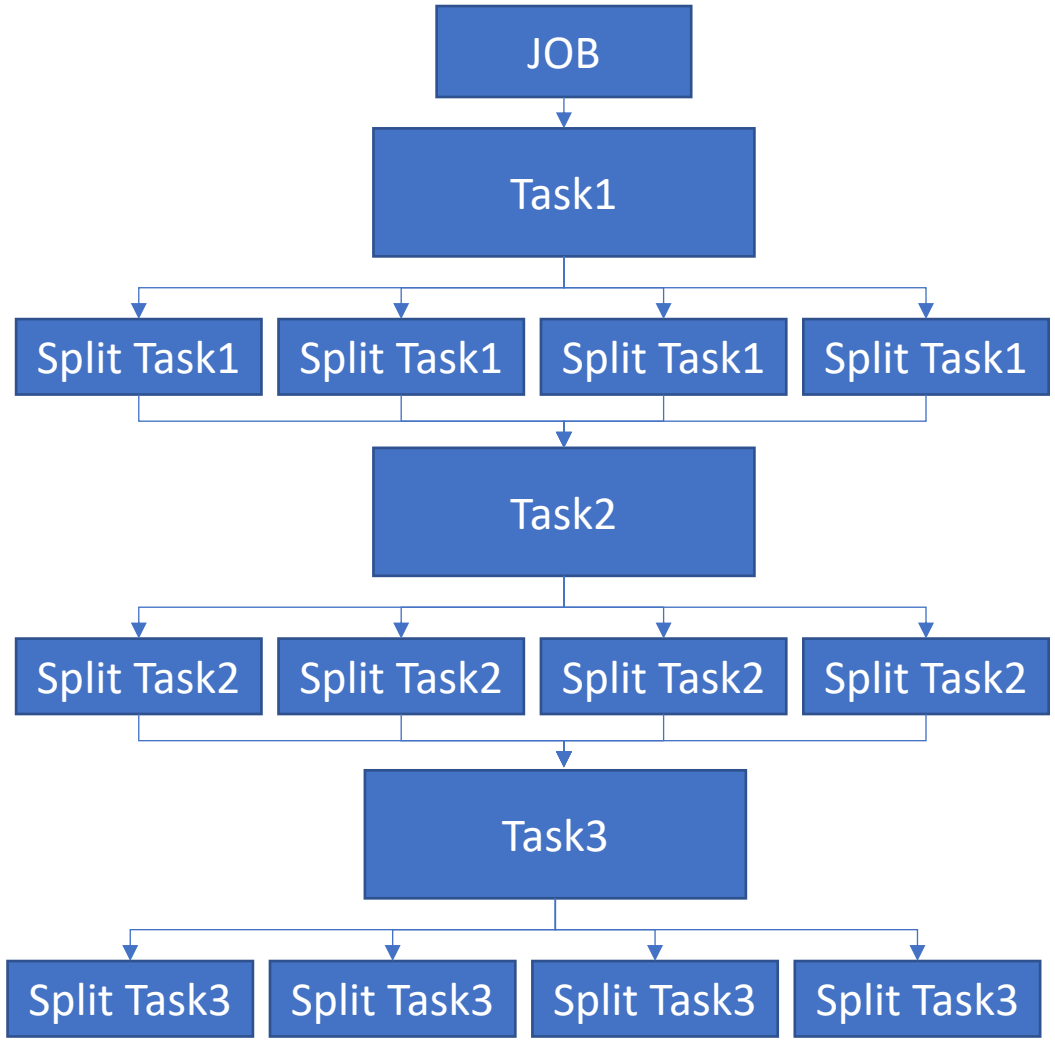
← 프로세스 풀 생성
(복수의 프로세스 생성)

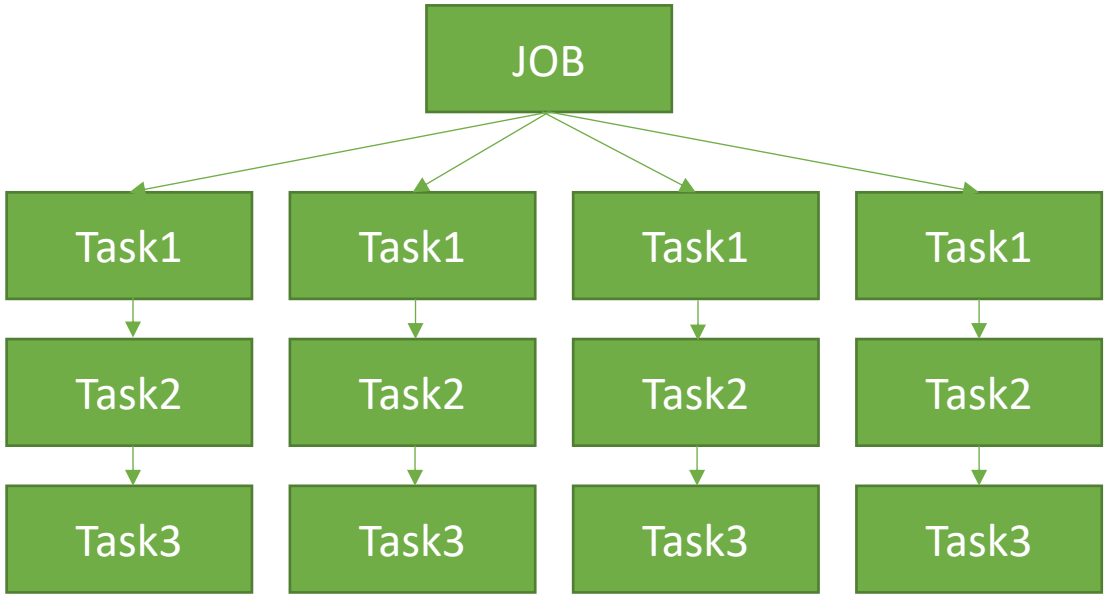
# Python Parmap 모듈을 활용한 병렬화

- conda install parmap

```
import parmap
# You want to do:
mylist = [1,2,3]
argument1 = 3.14
argument2 = True
y = [myfunction(x, argument1, mykeyword=argument2) for x in mylist]
# In parallel:
y = parmap.map(myfunction, mylist, argument1, mykeyword=argument2)
```

# Fine-grained 병렬화 vs Coarse-grained 병렬화

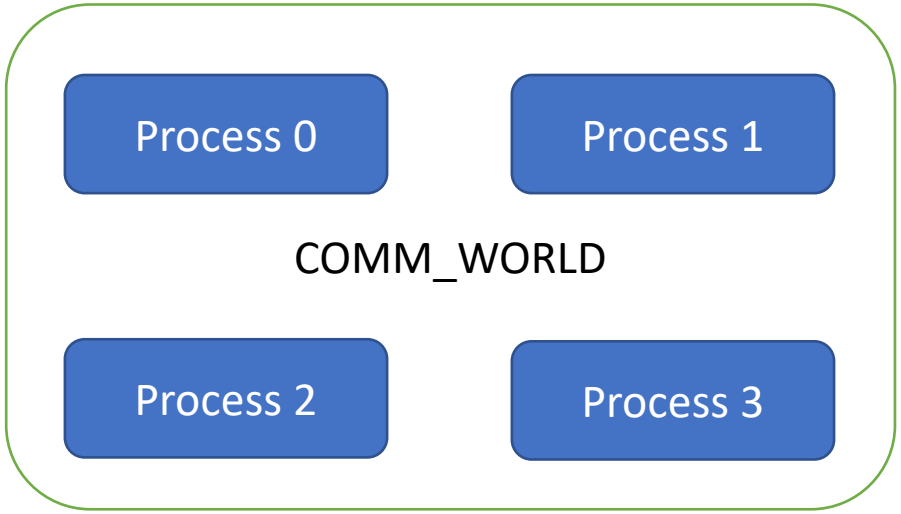


# MPI

```
from mpi4py import MPI

mpicomm = MPI.COMM_WORLD
mpisize = mpicomm.Get_size()
mpirank = mpicomm.Get_rank()
```




- comm_world: 정보를 주고 받는 프로세스의 그룹
- MPI size: 해당 커뮤니케이션 그룹에서 프로세스의 갯수
- MPI rank: 해당 커뮤니케이션 그룹에서 각 프로세스의 일련 번호 (0번 부터 시작)

# MPI

```
from mpi4py import MPI

mpicomm = MPI.COMM_WORLD
mpisize = mpicomm.Get_size()
mpirank = mpicomm.Get_rank()
```

- mpicomm.bcast(*data*,*root=num*)
- mpicomm.send(*data*,*dest=num*)
- mpicomm.recv(*data*,*source=num*)
- mpicomm.scatter(*data*,*root=num*)
- mpicomm.gather(*data*,*root=num*)
- mpicomm.barrier()

- 대문자 명령어 (Bcast, Send, Recv..)와 소문자 명령어 (bcast, send, recv..)
  - 대문자 명령어 (low level 동작): 기본 데이터 타입만 지원 (float, int, character, …)
  - 소문자 명령어 (high level 동작): python 의 모든 데이터 타입을 지원 (picklable)

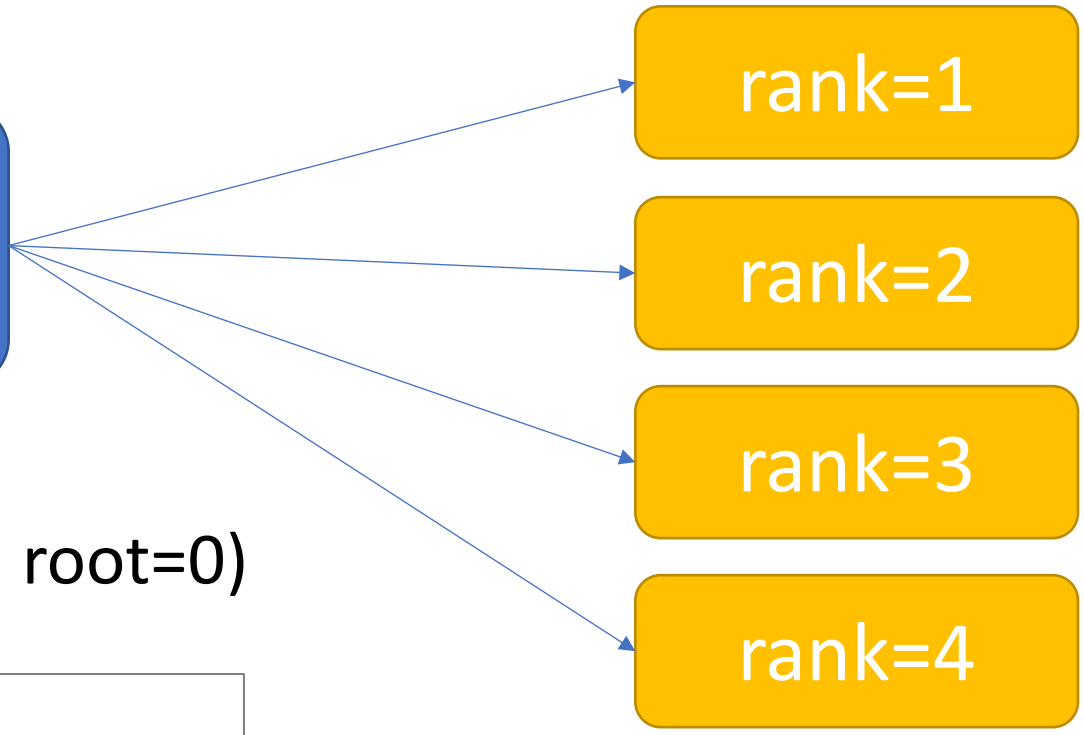
- Python subroutine 자체의 overhead가 있기 때문에 fine-grain의 parallelization은 일의 규모가 작을 때 오히려 느려질 수도 있다.

- 좀 더 추상적인 개념의 work를 설정하고 그 work를 나누어서 하는 것이 python level에서는 더 적절할 수 있다.

# MPI – bcast

data = [1, 2, 3, 4]

Root rank
(주로 rank=0)




data = [1, 2, 3, 4]

data = [1, 2, 3, 4]

data = [1, 2, 3, 4]

data = [1, 2, 3, 4]

데이터 보내는 쪽 명령

mpicomm.bcast(data, root=0)

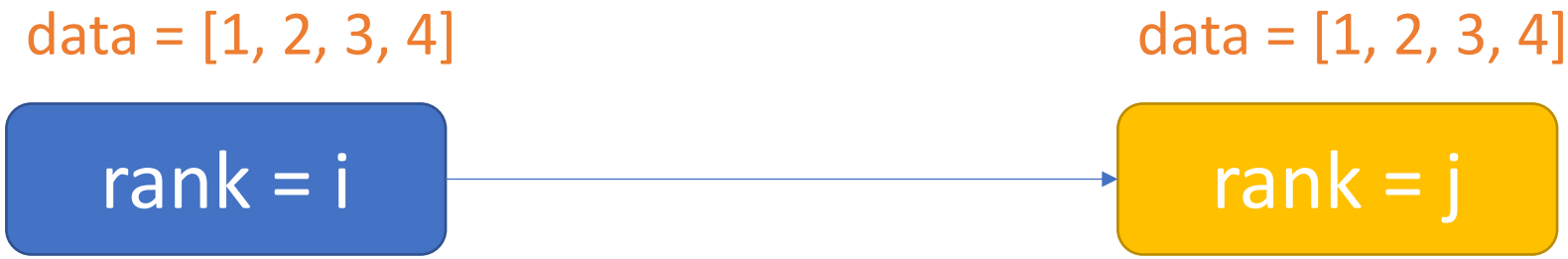
데이터 받는 쪽 명령

data = mpicomm.bcast(None, root=0)

*Dummy data*

```
# 참고: low-level Bcast 용례
if mpirank == 0:
    data = np.arange(10,dtype=int)
else:
    data = np.empty(10,dtype=int)
#mpicomm.Bcast([data,MPI.INT],root=0)
mpicomm.Bcast(data,root=0)
```

# MPI – send / recv




데이터 보내는 쪽 명령

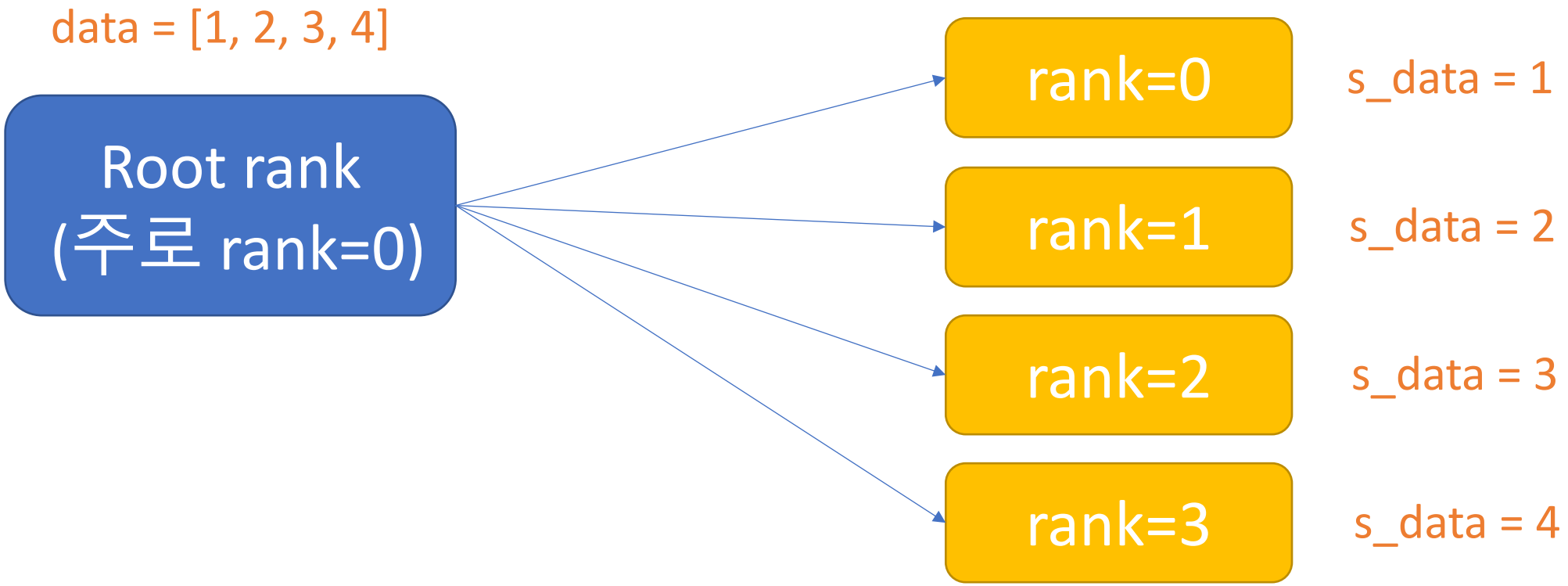
mpicomm.send(data, dest=j, tag=tag)

데이터 받는 쪽 명령

data = mpicomm.recv(source=i, tag=tag)

- tag: 선택적으로 주어지는 정보로서 주고 받는 대상이 일치하는지 여부를 확인하기 위해 사용 (즉, tag가 같아야 함, 정수값)
- MPI.ANY: dest 또는 source 에 사용 가능, 특정 rank를 지정하지 않을 때 사용

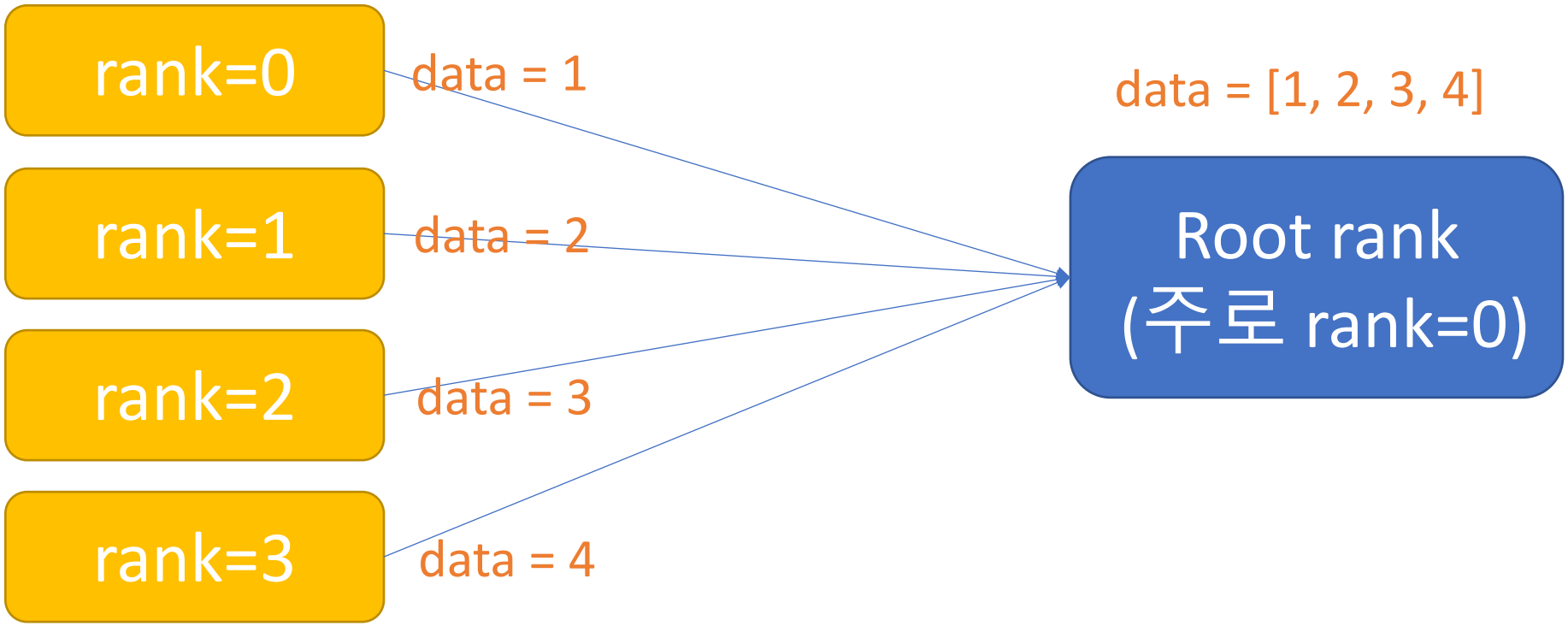
# MPI – scatter




| | bcast | scatter |
|---|---|---|
| 보내는 프로세서 포함 | x | o |
| 데이터 나눔 (split) | x | o |

```
s_data = mpicomm.scatter(data, root=0)
```

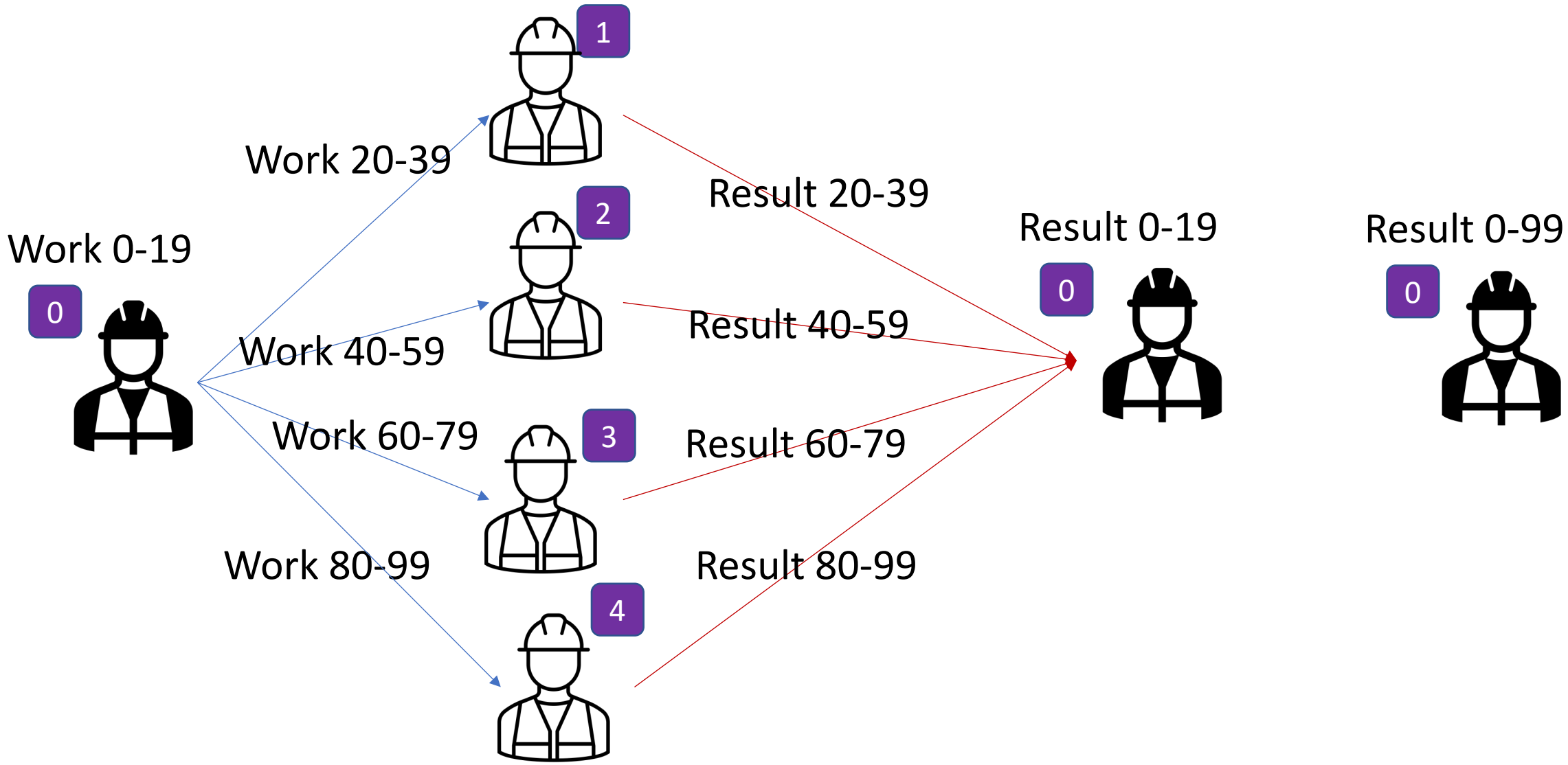
# MPI – gather (allgather)




```
data_gathered = mpicomm.gather(data, root=0)
```

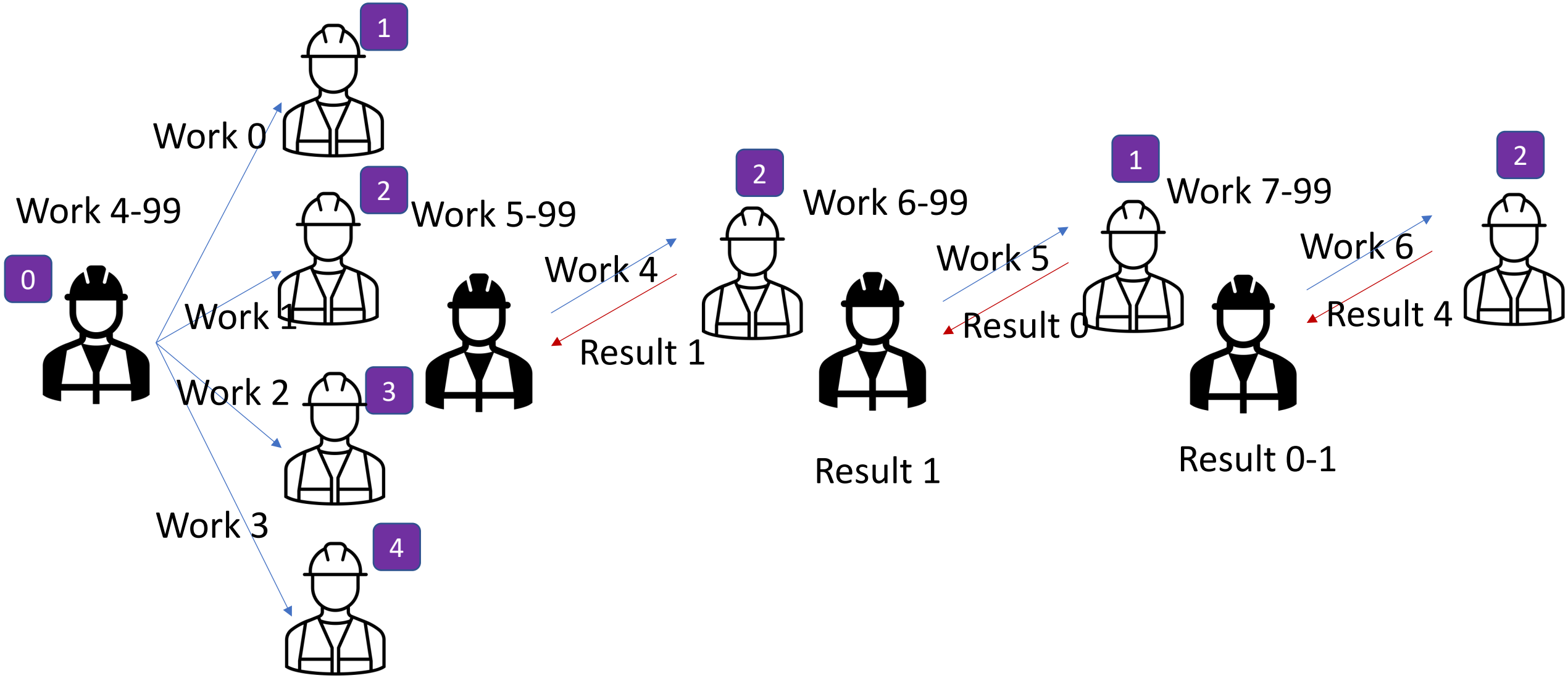
# Equally Distributed Works

나눠서 수행할 **일의 계산량이 일정할 때** 적합하다. 주로 fine-grained



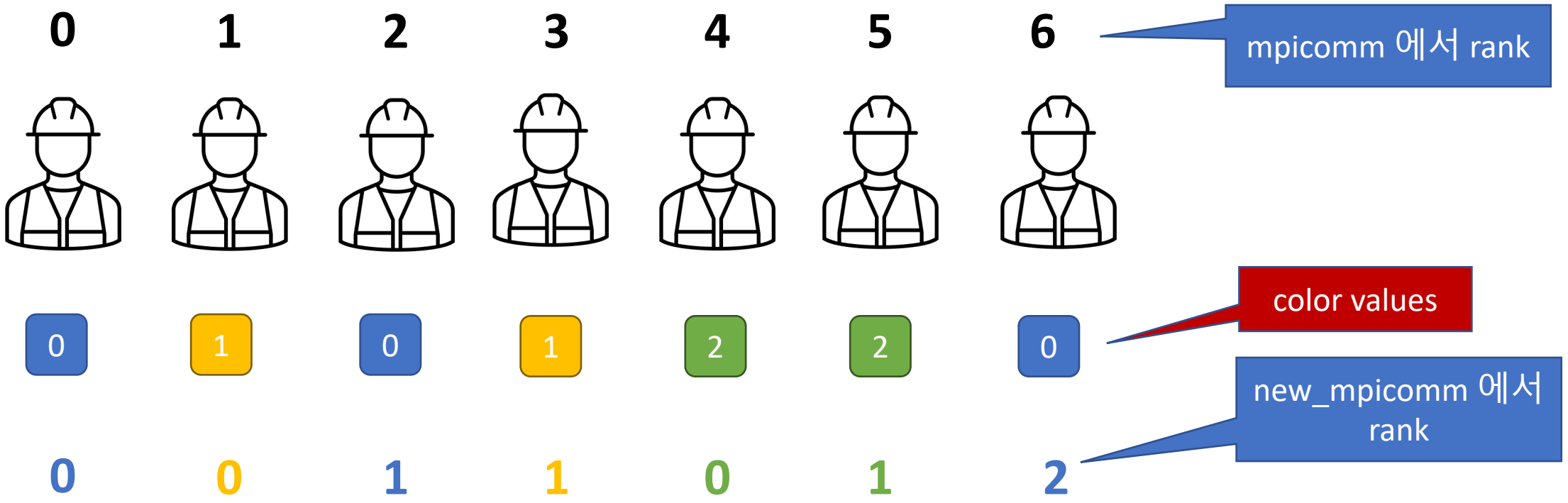
# Master / Slave Parallelization

나눠서 수행할 **일의 계산량이 일정하지 않을 때** 적합하다. 주로 coarse-grained



# COMM_WORLD 나누기

- new_mpicomm = mpicomm.Split(*color*,*key*)
  - color: 같은 communicator에 들어갈 프로세서들은 같은 값을 준다.
  - key: 같은 communicator에서 rank 순서를 정하기 위한 값 (보통 mpirank를 쓰면 됨)



# MPI Tips

- 소문자 MPI 명령어 (bcast, send, recv, … )등으로 작동하는 코드를 우선 만든다.
- 처음부터 대문자 명령 (Fortran/C 스타일)으로 하는 것은 좋지 않다. Fortran/C 에서와는 달리 **Python에서는 임의의 object를 자유롭게 주고 받는 것이 쉽다는 점을 활용하자!**
- 단, 주고 받는 데이터는 pickling이 가능한 데이터 형태로 사용한다. Pickling이 어려운 객체를 주고 받을 때는 객체의 중요 정보만 전달하고 수신 측에서 해당 정보를 이용해서 객체를 새로 생성하는 방법을 사용할 수 있다.
- 필요에 따라 프로파일링을 해 보고 대문자 명령으로 바꾸는 것을 고려할 수 있다. (하지만 coarse-grained 병렬화에서는 속도 차이가 거의 없을 수도 있다.)
- Block send/recv를 **immediate (non-blocking) send/recv** 등 (isend/irecv) 으로 바꾸는 것을 고려 할 수 있다. (의도하지 않은 결과가 될 수도 있다. 디버깅 필요!)

# Cluster Queueing 시스템을 활용한 병렬화

Queueing 시스템: Torque/PBS, Slurm

PBS 작업 스크립트의 예시

```
#!/bin/sh

#PBS -N name_of_the_job
#PBS -o stdout.log
#PBS -j oe
#PBS -l walltime=24:00:00
#PBS -l ncpus=12

NCPU=`wc -l < $PBS_NODEFILE`
mpirun -np $NCPU your_mpi_job
mpirun -f $PBS_NODEFILE your_mpi_job
```

Slurm 작업 스크립트의 예시

```
#!/bin/sh
#SBATCH -J name_of_the_job
#SBATCH -N 1
#SBATCH -n 12
#SBATCH -o output.log
#SBATCH -e output.log

srun your_mpi_job
```

# Submitit

- https://github.com/facebookincubator/submitit
- Python 서브루틴을 Slurm job으로 만들어 주는 툴
- conda install -c conda-forge submitit