

# Parallel Programming in Python : a Pure-MPI code

Jiaxi Yu

Supervisor: Vincent Keller

Nov 06, 2020

# Contents

## ■ the Hybrid version:

- structure
- performance
- pros and cons

## ■ the Pure-MPI version

- structure
- performance
- pros and cons

# The Hybrid version

## ■ Structure:

```
# catalogue reading using ascii.read (2e8×5 array)

# generate N uniform random array (N=30, length 2e8)
def stat(**arg):
    # transfer a uniform array to Gaussian distributions
    # scatter a catalogue column with them
    # for each scattered array, calculate another quantity
    return quantity

def log_like(**argv2):
    # use multiprocessing.Pool for N times stat(**argv2)
    # model = np.mean(stat return, axis=0)
    # chi_square = ((data-model)/error)**2
    return -0.5*chi_square
```

MC\_Sampler(log\_like) with mpi4py

# The Hybrid version

## ■ Performance:

```
# catalogue reading (~4min) using ascii.read (2e8×5 array)

# generate N uniform random array (~1min) (N=30, length 2e8)
def stat (~16s) (**arg):
```

```
def log_like (**argv2):
    # multiprocessing.Pool(~2.5min for N=30)
```

```
MC_Sampler(log_like) with mpi4py(4.5hours for -N 16 -c 64)
```

# The Hybrid version

## ■ Pros:

1. easy to implement and read
2. no extra communication between processes for `stat()` and for `MC_Sampler()`

## ■ Cons:

1. `multiprocessing.Pool` has very long communication time
2. `multiprocessing.Pool` only requests processes in one nodes, so  $N < 31$  and more nodes don't improve its speed

## ■ Expectations for MPI

1. compute N Gaussian random arrays faster
2. number of random arrays N can be larger than 31
3. faster sampling

# The Pure-MPI version

## ■ Structure: Parent

```
# initialise MPI
# catalogue reading using hdf5

def model(**arg):
    # Spawn N children processes
    # broadcast array size and scatter parameters to them
    # reduce the sum of all children to the root process
    return sum/N

def log_like(**argv2):
    return -0.5*((data-model(**argv2))/error)**2

MC_Sampler(log_like) without mpi4py
```

# The Pure-MPI version

## ■ Structure: Children

```
# initialise mpi4py
get parent()
receive the broadcast info
```

```
# generate a uniform random array in each parent process
# transfer this uniform array to Gaussian distributions
# scatter a catalogue column with them
# for each scattered array, calculate another quantity

reduce the sum to the root process
```

# The Pure-MPI version

## ■ Performance tests:

Login nodes:

Cori, 32 physical cores and 2 Threads

Computing nodes:

Haswell, 32 physical cores and 2 Threads

Memory limit:

Cori 64G, Haswell 128G

Test setting:

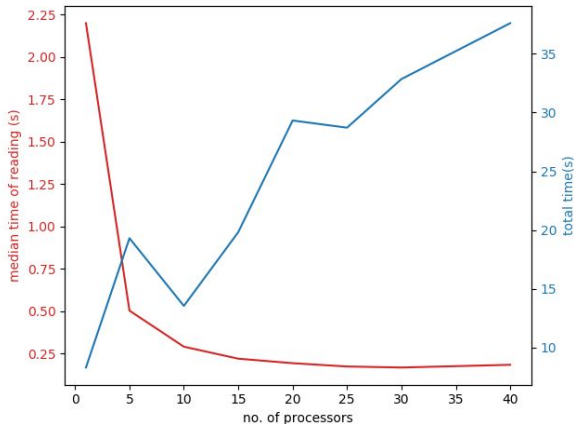
```
salloc -N 2 -n 128 -c 1 -q interactive -C Haswell
```



# The Pure-MPI version

## ■ Performance tests:

# reading catalogue using hdf5:



data will gather in the  
end of reading



long communication time  
& no enough memory

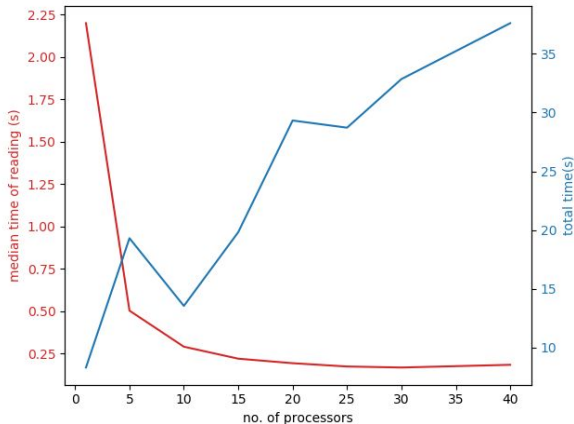


no need to parallelise  
(still much better than  
4-min ascii reading)

# The Pure-MPI version

## ■ Performance tests:

# reading catalogue using hdf5:



data will gather in the  
end of reading



long communication time  
& no enough memory



no need to parallelise

\* `mpiexec -n 64:` 74s

\* `mpiexec -n 128:` >370s

# The Pure-MPI version

## ■ Performance tests:

```
# parent-child structure tests
x = np.linspace(0.,101.,int(8e7))
maxproc = #
def log_like (argv):
    model = par[0]*x+par[1]
    comm = MPI.COMM_SELF.Spawn (sys.executable,
                                args=[' child.py'],maxprocs=maxproc)
    lenx = np.array(len(x), 'i')
    sigma = np.array(1., 'd')
    comm.Bcast ([lenx, MPI.INT], root=MPI.ROOT)
    comm.Bcast ([sigma, MPI.DOUBLE], root=MPI.ROOT)
    noise = np.zeros(lenx, 'd')
    comm.Reduce (None, [noise, MPI.DOUBLE], op=MPI.SUM, root=MPI.ROOT)
    y_mean = 2*x+3+noise/maxproc
    comm.Disconnect ()
    return -0.5*((y_mean-model)**2/error**2).sum()

print(log_like([a,b]))
```

# The Pure-MPI version

## ■ Performance tests:

```
# parent-child structure tests
'child.py'

# initialise mpi4py
lenx      = np.array(0., 'i')
sigma     = np.array(0., 'd')
comm.Bcast([lenx, MPI.INT], root=0)
comm.Bcast([sigma, MPI.DOUBLE], root=0)

# make sure the Gaussian random is stable
uniform = np.random.RandomState(seed=rank+1).rand(lenx)
half    = int(lenx/2)
noise   =
append(sigma*sqrt(-2*log(uniform[:half]))*cos(2*pi*uniform[half:]))
,\
sigma*sqrt(-2*log(uniform[:half]))*sin(2*pi*uniform[half:]))

comm.Reduce([noise, MPI.DOUBLE], None, op=MPI.SUM, root=0)
comm.Disconnect()
```

# The Pure-MPI version

## ■ Performance tests:

```
# parent-child structure tests
```

```
x = np.linspace(0.,101.,int(8e7)) # close to the upper memory  
limit
```

**maxproc (number of children processes)**

```
> mpiexec -n Ntask python parent.py (Ntask parent processes)
```

```
# baseline 1
```

```
replace children with multiprocessing.Pool with 30 processes
```

```
# baseline 2
```

```
Nnodes = 2, maxproc = 30, Ntasks = 2
```

# The Pure-MPI version

## ■ Performance tests:

# parent-child structure tests

	<b>Nnode</b>	<b>ntask</b>	<b>maxproc</b>	<b>time(s)</b>
<b>base-Hybrid</b>	2	2	30	145.0
<b>base-MPI</b>	2	2	30	37.05
<b>MPI</b>	4	2	60	46.44
<b>MPI</b>	4	3	30	36.27

# The Pure-MPI version

## ■ Performance tests:

# likelihood combined with the Monte-Carlo Sampler

(Problematic!)

1. **Pymultinest:** cannot work at all even though its own mpi4py is disabled
2. **emcee:** a sequential code. But it stops after 17% points are sampled (no error reports).

# The Pure-MPI version

## ■ Pros:

1. maxproc can be larger than 30
2. Gaussian array has been generated 4 times faster than the hybrid version

## ■ Cons:

1. difficult to adjust to the external library
2. easily out of memory given the large data set, so data-preprocessing needed

## ■ Code&results:

[https://github.com/Jiaxi-Yu/parallel\\_programming\\_project.git](https://github.com/Jiaxi-Yu/parallel_programming_project.git)



**Thanks!**

# The Hybrid version:C

Self	ELF Object
57.82	libm-2.17.so
23.12	libc-2.17.so
18.84	MTSHAM
0.21	libpthread-2.17.so
0.00	ld-2.17.so
0.00	libnuma.so.1.0.0
0.00	libgfortran.so.3.0.0

Incl.	Self	Called	Function	Location
100.00	0.00	1	0x0000000000402c27	MTSHAM
100.00	0.00	1	main	MTSHAM
74.84	0.00	1	save_best	MTSHAM
74.84	0.00	1	best_fit	MTSHAM
74.53	3.93	102	apply_sham	MTSHAM
70.56	9.93	17 833 783 221	mt19937_get_gauss	MTSHAM
25.16	0.00	1	read_cata	MTSHAM
25.16	0.87	1	read_ascii_data	MTSHAM
2.81	2.81	8 916 891 611	mt19937_get	MTSHAM
0.50	0.50	881 673 180	ast_set_var	MTSHAM
0.50	0.39	881 673 180	ast_eval	MTSHAM
0.30	0.00	101	comp_2pcf_auto	MTSHAM
0.21	0.00	101	kdtree_count_auto_pairs	MTSHAM
0.21	0.00	101	kdtree_dual_auto_intbi...	MTSHAM
0.21	0.21	28 220 463	kdtree_dual_auto_intbi...	MTSHAM
0.11	0.11	881 673 180	ast_eval_double.part.3	MTSHAM
0.09	0.01	101	kdtree_build	MTSHAM
0.08	0.08	1 654 582	kdtree_build'2	MTSHAM
0.01	0.01	93 738 432	compare_mass	MTSHAM

C-profiling costs 6 hours  
& gprof didn't work  
& data all in ascii format

# The Pure-MPI version

## ■ Structure:

failed due to unnoticed  
deadlock

```
# generate N uniform random array
def stat(**arg):
    # transfer N uniform array to Gaussian distributions
    # scatter a catalogue column with them
    # for each scattered array, calculate another quantity
    return mean(N quantities)

def log_like(**argv2):
    # use MPI.Pool for stat(**argv2)
    # chi_square = ((data-model)/error)**2
    return -0.5*chi_square

a serial MC_Sampler(log_like)
```