

Docker封装的分布式矩阵运算

这是一个使用Python3.6编写的简单的分布式矩阵计算的测试程序，支持分布式矩阵乘法与分布式矩阵求逆。整个项目使用Docker封装，容易批量配置。矩阵乘法使用分块乘法实现分布式，矩阵求逆使用分块消元算法实现分布式，仅仅实现功能，在网络传输、性能与并行度上依旧有所不足。

- `multiprocessing` 实现分布式的任务分配
- `flask` 实现Web控制界面
- `h5py` 实现结果HDF5格式存储

已发布到Docker Hub：https://hub.docker.com/r/wnjxyk/simple_distributed_matrix，可以在这个网址拉取镜像并且测试。

Github链接：<https://github.com/WNJXYK/Distributed-Matrix-Calculation>

博客链接：<https://blog.wnjxyk.cn/2018/12/29/docker封装的简易分布式矩阵计算/>

使用与测试

Step One : 从镜像新建Docker容器

首先从镜像仓库拉取本镜像。

```
docker pull wnjxyk/simple_distributed_matrix
```

然后从镜像新建了5容器，分别映射端口号为8080、8081~8084。并且将8081~8084端口号容器的CPU性能限制在1%。这里8080作为主控节点，8081~8084作为分布式计算节点。

限制CPU性能是为了最大化所演示的分布式计算的效果，因为程序比较简单，并未对于网络传输与性能有所优化，所以中心的控制节点会成为性能的瓶颈。我们将控制节点的性能不设限制，计算节点的性能限制在1%，可以展现不考虑中心节点与网络传输限制的理想情况下，分布式计算的效果。

```
docker run -d -it -p 8080:80 matrix python
/root/Distributed_Matrix_Method/Distributed.py
docker run -d -it -p 8081:80 --cpus=0.01 matrix python
/root/Distributed_Matrix_Method/Distributed.py
docker run -d -it -p 8082:80 --cpus=0.01 matrix python
/root/Distributed_Matrix_Method/Distributed.py
docker run -d -it -p 8083:80 --cpus=0.01 matrix python
/root/Distributed_Matrix_Method/Distributed.py
docker run -d -it -p 8084:80 --cpus=0.01 matrix python
/root/Distributed_Matrix_Method/Distributed.py
```

Step Two : 测试性能

查明8080端口Docker容器在局域网中的IP，然后在Docker宿主浏览器中打开这五个Docker控制页面，进行分布式计算性能测试。

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
fbe9b3dc6549	matrix	"python /root/Distru..."	7 minutes ago	Up 7 minutes	0.0.0.0:8084->80/tcp	confident_benz
0f24af045e91	matrix	"python /root/Distru..."	7 minutes ago	Up 7 minutes	0.0.0.0:8083->80/tcp	stupefied_kirch
dead81672463	matrix	"python /root/Distru..."	7 minutes ago	Up 7 minutes	0.0.0.0:8082->80/tcp	epic_mahavira
b5b8972a5cf9	matrix	"python /root/Distru..."	7 minutes ago	Up 7 minutes	0.0.0.0:8081->80/tcp	compassionate_poitras
6cadd7e58733	matrix	"python /root/Distru..."	7 minutes ago	Up 7 minutes	0.0.0.0:8080->80/tcp	infallible_mestorf

这里我们的控制节点名字叫做 `infallible_mestorf`，四个计算节点的名字叫做 `confident_benz`、`stupefied_kirch`、`epic_mahavira` 与 `compassionate_poitras`。

首先，我们测试一下只使用一个计算节点的情况下做矩阵乘法，这里我们启用计算节点 `compassionate_poitras`。

Distributed_Matrix_Method — docker stats — 156x24							
CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
fbe9b3dc6549	confident_benz	0.11%	35.66MiB / 1.952GiB	1.78%	14.2kB / 43kB	0B / 0B	4
0f24af045e91	stupefied_kirch	0.05%	33.86MiB / 1.952GiB	1.69%	1.13kB / 0B	0B / 0B	4
dead81672463	epic_mahavira	0.07%	33.89MiB / 1.952GiB	1.70%	1.13kB / 0B	0B / 0B	4
b5b8972a5cf9	compassionate_poitras	1.04%	41.39MiB / 1.952GiB	2.07%	10.7MB / 4.88MB	0B / 0B	6
6cadd7e58733	infallible_mestorf	4.20%	199.9MiB / 1.952GiB	10.00%	5.85MB / 11.4MB	0B / 0B	11

在控制页面中，我们可以看到这次矩阵乘法使用了61.56秒。

Host

Address: 0.0.0.0 : 51234

Authenticate: Matrix_Method

Running

Terminate

Worker

Address: 1270.0.1 : 51234

Authenticate: Matrix_Method

Stopped

Start

Task

Matrix Size: 1024Block Size: 256

Inverse Task

Multiply Task

Type: Mul

Status: Correct(Saved)

Epoch: 1/1

Unit: 64/64

Time: 61.55 S

Download HDF5 File

然后，我们令四个计算节点一起工作计算同样规模的矩阵乘法。

Distrubuted_Matrix_Method — docker stats — 156x24									
CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS		
fbe9b3dc6549	confident_benz	1.13%	41.39MiB / 1.95GiB	2.07%	5.38MB / 2.27MB	0B / 0B	6		
0f24af045e91	stupefied_kirch	1.16%	39.34MiB / 1.95GiB	1.97%	5.35MB / 2.74MB	0B / 0B	6		
dead81672463	epic_mahavira	1.10%	40.69MiB / 1.95GiB	2.04%	6.44MB / 3.27MB	0B / 0B	6		
b5b8972a5cf9	compassionate_poitras	1.07%	41.52MiB / 1.95GiB	2.08%	77.3MB / 38.3MB	0B / 0B	6		
6cadd7e58733	infallible_mestorf	8.02%	212.7MiB / 1.95GiB	10.64%	47.2MB / 95.5MB	0B / 0B	15		

在控制页面中，我们可以看到这次矩阵乘法使用了19.86秒，提升非常大。

Host

Address: 0.0.0.0 : 51234

Authenticate: Matrix_Method

Running Terminate

Worker

Address: 127.0.0.1 : 51234

Authenticate: Matrix_Method

Stopped Start

Task

Matrix Size: 1024 Block Size: 256

Inverse Task Multiply Task

Type: Mul Status: Correct(Saved)

Epoch: 1/1 Unit: 64/64 Time: 19.86 S

Download HDF5 File

再测试一下矩阵求逆运算，可以看到单个计算节点需要使用100秒，而四个计算节点只需要使用约26秒。

Host

Address: 0.0.0.0 : 51234

Authenticate: Matrix_Method

Running Terminate

Worker

Address: 172.17.0.2 : 51234

Authenticate: Matrix_Method

Stopped Start

Task

Matrix Size: 1024 Block Size: 256

Inverse Task Multiply Task

Type: Inv Status: Finished(Saved)

Epoch: 4/4 Unit: 16/16 Time: 100.07 S

Download HDF5 File

Host

Address:0.0.0.0:51234

Authenticate:Matrix_Method

Running

Terminate

Worker

Address:127.0.0.1:51234

Authenticate:Matrix_Method

Stopped

Start

Task

Matrix Size:1024Block Size:256

Inverse Task

Multiply Task

Type:Inv

Status:Correct(Saved)

Epoch:4/4

Unit:16/16

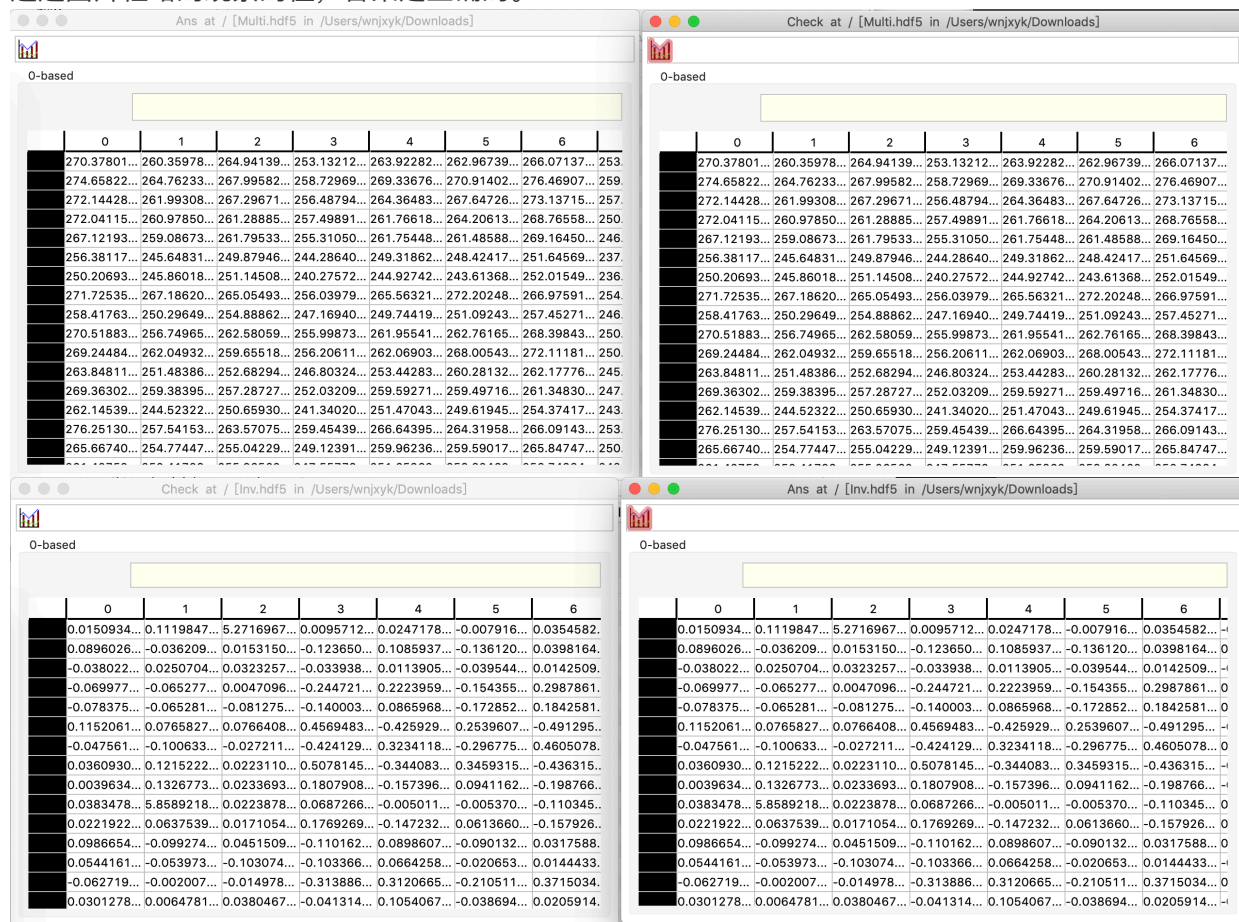
Time:26.51 S

Download HDF5 File

Step Three : 正确性测试

控制面板会自动测试结果的正确性，如果显示结果为 `Correct` 就可以判断结果是正确的。同时系统会讲结果存储为一个HDF5格式的文件，可以点击最下面的按钮进行下载，然后使用HDF View进行浏览与检验。

其中 `Ans` 数据集是我们分布式计算出来的答案，`Check` 数据集是使用Numpy计算出来的答案，可以通过图片粗略的观察的值，答案是正确的。



分布式实现

乘法与求逆元都使用了分块的思路，将一个大小为 $size$ 的矩阵分成大小为 blk 的分块 $Block_{i,j}$ ，那么会得到一个 $bkc = \frac{size}{blk}$ 的方阵，其每个元素都是一个 blk 的方阵。

分块矩阵乘法

那么原来矩阵乘法 $Ans = A \cdot B$ ，可以转化成 $BlockAns_{i,j} = \sum_{k=0}^{bkc} BlockA_{i,k} \cdot BlockB_{k,j}$ 。可以看出分块之后的乘法之间是完全独立的，所以我们可以把这些耗时的矩阵乘法任务分发给计算节点运算，然后将结果返回加入结果的分块矩阵中就可以了。假设计算节点有 CNT 个，我们可以把原本计算矩阵乘法的复杂度从 $O(n^3)$ 变换为 $O(blk^3 \times (\lceil \frac{bkc}{CNT} \rceil))$ 。

Sequential Block-Based Gauss-Jordan Algorithm

矩阵求逆的方法使用的还是消元，不过使用了一种序列分块版本，对分布式运算更加友好。这是一种将高斯消元的过程分块的算法，它的好处分块的过程中，块与块之间会存在很多不相关的乘法运算，利用这个性质，我们就可以将不相关的运算分散到运算节点上去。假设计算节点有 CNT 个，我们可以以原本 $O(n^3)$ 的复杂度降低为 $O(blk^3 \times bkc \cdot (\lceil \frac{bkc}{CNT} \rceil + \lceil \frac{bkc(bkc-1)}{CNT} \rceil))$

Docker常用指令

1. 从Docker容器创建镜像

```
docker commit <container ID> name:tag
```

2. 查看Docker之间的网络组织情况

```
docker network inspect bridge
```

3. 统计Docker容器使用情况

```
docker stats # CPU使用情况  
docker ps # 运行中容器列表  
docker ps -a # 所有容器列表
```

4. 关闭所有Docker执行进程

```
docker stop $(docker ps -a | awk '{ print $1}' | tail -n +2)
```

5. 关闭所有Docker容器

```
docker rm $(docker ps -a | awk '{ print $1}' | tail -n +2)
```