一、使用cgroup配置NUMA

1. 查看当前服务器NUMA节点的配置

```
numactl -H       #查看NUMA节点的配置
```

```
zmq@ubuntu:/$ numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
node 0 size: 63937 MB
node 0 free: 22914 MB
node 1 cpus: 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
node 1 size: 64485 MB
node 1 free: 112 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
```

2. 查看NUMA节点的状态

```
numastat -m      #查看NUMA节点的状态
```

3. 安装cgroups工具

```
sudo apt-get install cgroup-tools
```

4. 创建cgroup

```
sudo cgcreate -g memory:/numa0
sudo cgcreate -g memory:/numa1
sudo cgcreate -g cpuset:/numa0
sudo cgcreate -g cpuset:/numa1
```

5. 设置内存限制，节点0内存大小为64G，节点1内存大小为8G

```
sudo cgset -r memory.limit_in_bytes=$((64 * 1024 * 1024 * 1024)) numa0
sudo cgset -r memory.limit_in_bytes=$((8 * 1024 * 1024 * 1024)) numa1
```

6. 绑定NUMA节点0和1到相应的cgroup

```
sudo cgset -r cpuset.mems=0 numa0        # cpuset.mems=0 指NUMA节点0的内存
sudo cgset -r cpuset.mems=1 numa1        # cpuset.mems=1 指NUMA节点1的内存
```

7. 绑定CPU到相应的NUMA节点，与第一步中查看到的相对应

```
sudo cgset -r
cpuset.cpus=0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38 numa0
sudo cgset -r
cpuset.cpus=1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39 numa1
```

8. 查看cgroups状态

```
lscgroup | grep "numa"       #查找创建的cgroup
cat /sys/fs/cgroup/memory/numa0/memory.stat      #查看numa0的内存状态
cat /sys/fs/cgroup/cpuset/numa0/cpuset.cpus      #查看numa0绑定的cpu
cat /sys/fs/cgroup/memory/numa1/memory.stat      #查看numa1的内存状态
cat /sys/fs/cgroup/cpuset/numa2/cpuset.cpus      #查看numa1绑定的cpu
cgget -r memory.usage_in_bytes numa0             #查看numa0当前的内存使用情况
cgget -r memory.usage_in_bytes numa1             #查看numa1当前的内存使用情况
```

9. 运行进程

使用 `cgexec` 将进程绑定到相对应的cgroup，并结合 `numactl` 命令来控制内存的分配。

```
sudo cgexec -g memory,cpuset:/numa0 numactl --membind=0,1 --cpunodebind=0
your_command      #在numa节点0上运行进程,依次分配使用NUMA0和NUMA1的内存资源
sudo cgexec -g memory,cpuset:/numa0 numactl --interleave=0,1 --cpunodebind=0
your_command       #在numa节点0上运行进程,交错分配使用NUMA0和NUMA1的内存资源
```

10. 删除cgroup

```
sudo cgdelete -g memory,cpuset:/numa0       #删除numa0
sudo cgdelete -g memory,cpuset:/numa1       #删除numa1
# sudo cgdelete memory:/numa0
# sudo cgdelete cpuset:/numa0
# sudo cgdelete memory:/numa1
# sudo cgdelete cpuset:/numa1
```

二、测试延迟和带宽

1. 按 `一、9` 所示分别测试访问当前NUMA节点内存和访问跨NUMA节点内存的平均访问延迟和带宽，使用工具 Intel MLC

```
# 临时禁用自动 NUMA 平衡
echo 0 | sudo tee /proc/sys/kernel/numa_balancing
# 恢复自动 NUMA 平衡
echo 1 | sudo tee /proc/sys/kernel/numa_balancing

# 每次执行前先清空缓存
# 仅清除页面缓存
# sudo sync; echo 1 | sudo tee /proc/sys/vm/drop_caches
# 清除目录项和inode
# sudo sync; echo 2 | sudo tee /proc/sys/vm/drop_caches
# 清除页面缓存，目录项和inode
sudo sync;echo 3 | sudo tee /proc/sys/vm/drop_caches
# 在numa节点0上运行进程,使用NUMA0的内存资源
sudo cgexec -g cpuset:/numa0 -g memory:/numa0 ./latency_and_bandwidth_test
# 清除页面缓存，目录项和inode
sudo sync;echo 3 | sudo tee /proc/sys/vm/drop_caches
# 在numa节点0上运行进程,使用NUMA1的内存资源
sudo cgexec -g cpuset:/numa0 -g memory:/numa1 ./latency_and_bandwidth_test
```

2. 存在的问题：

1. 使用cgroup进行tinymembench的测试时两个没区别

```
sudo cgexec -g cpuset:/numa0 -g memory:/numa1 ./tinymembench
```

使用numactl进行测试时两个有区别

```
numactl --cpubind=0 --membind=1 ./tinymembench
```

原因：系统默认使用cpu同节点上的内存，cgroup指定cpu后只会在cpu节点上分配内存。

2. 现在的模拟方法使用跨节点的内存访问来模拟HBM，虽然获得了更高的延迟，但带宽也更低了。

解决方法：在NUMA0上插奇数根内存条，在NUMA1上插偶数根内存条，由于二路内存交错编址，NUMA0访问NUMA1中的内存带宽会比访问NUMA0中的内存带宽高，但由于只能实现二路交错编址，模拟的HBM带宽只能是DRAM的两倍。

3. 用numactl控制在哪个节点的cpu和内存上运行，用cgroup控制内存大小可行。

```
numactl --cpubind=0 --membind=1 sudo cgexec -g memory:/numa1
./latency_and_bandwidth_test
```

4.
```
# 使用 --interleave 选项可以在所有可用的NUMA节点之间交替分配内存：
numactl --interleave=all ./your_program
# 或者只在特定的节点间交替分配内存（例如节点0和1）：
numactl --interleave=0,1 ./your_program
# 使用 --membind 选项可以绑定内存到特定的NUMA节点：
numactl --membind=0 ./your_program
# 使用 --preferred 选项可以设置首选的NUMA节点，但允许其他节点分配内存：
numactl --preferred=0 ./your_program
```

mem0 1根 32G mem1 2根 共64G

mlc结果

```
zmq@ubuntu:~/MLC$ ./mlc
Intel(R) Memory Latency Checker - v3.11a
*** Unable to modify prefetchers (try executing 'modprobe msr')
*** So, enabling random access for latency measurements
Measuring idle latencies for random access (in ns)...
                Numa node
Numa node            0       1
       0          94.8   158.1
       1         158.3    93.3

Measuring Peak Injection Memory Bandwidths for the system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using traffic with the following read-write ratios
ALL Reads        :      51995.7
3:1 Reads-Writes :      49062.5
2:1 Reads-Writes :      48396.0
1:1 Reads-Writes :      46795.9
Stream-triad like:      44072.3

Measuring Memory Bandwidths between nodes within system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using Read-only traffic type
                Numa node
Numa node            0       1
       0        17354.8 29850.7
       1        16308.4 34689.9
```

mem0 1根32G mem1 3根 共96G

mlc结果

```
zmq@ubuntu:~/MLC$ ./mlc | tee numa0_1_numa1_3_result
Intel(R) Memory Latency Checker - v3.11a
*** Unable to modify prefetchers (try executing 'modprobe msr')
*** So, enabling random access for latency measurements
Measuring idle latencies for random access (in ns)...
                Numa node
Numa node            0       1
       0          94.6   159.5
       1         158.7    95.6

Measuring Peak Injection Memory Bandwidths for the system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using traffic with the following read-write ratios
ALL Reads        :      34517.0
3:1 Reads-Writes :      32548.3
2:1 Reads-Writes :      31978.0
1:1 Reads-Writes :      31220.8
Stream-triad like:      29510.5

Measuring Memory Bandwidths between nodes within system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using Read-only traffic type
                Numa node
Numa node            0       1
       0        17363.0 16306.6
       1        16313.8 17371.1
```

mem0 1根32G mem1 8根 共256G

mlc结果

```
zmq@ubuntu:~/MLC$ ./mlc
Intel(R) Memory Latency Checker - v3.11a
*** Unable to modify prefetchers (try executing 'modprobe msr')
*** So, enabling random access for latency measurements
Measuring idle latencies for random access (in ns)...
                Numa node
Numa node            0       1
       0          97.1   159.6
       1         159.1    97.6

Measuring Peak Injection Memory Bandwidths for the system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using traffic with the following read-write ratios
ALL Reads        :      52208.0
3:1 Reads-Writes :      49429.3
2:1 Reads-Writes :      48788.6
1:1 Reads-Writes :      47113.2
Stream-triad like:      44896.0

Measuring Memory Bandwidths between nodes within system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using Read-only traffic type
                Numa node
Numa node            0       1
       0        17373.6 30580.1
       1        16319.2 35059.5
```

mem0 1根 mem1 8根 同节点和跨节点的最大带宽测试

```
zmq@ubuntu:~/MLC$ cat 00_numa0_1_numa1_8
Intel(R) Memory Latency Checker - v3.11a
Command line parameters: --max_bandwidth

Using buffer size of 100.000MiB/thread for reads and an additional 100.000MiB/thread for writes
*** Unable to modify prefetchers (try executing 'modprobe msr')
*** So, enabling random access for latency measurements

Measuring Maximum Memory Bandwidths for the system
Will take several minutes to complete as multiple injection rates will be tried to get the best bandwidth
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using traffic with the following read-write ratios
ALL Reads        :      17271.68
3:1 Reads-Writes :      14611.51
2:1 Reads-Writes :      13976.72
1:1 Reads-Writes :      11361.73
Stream-triad like:      13670.58

zmq@ubuntu:~/MLC$ cat 01_numa0_1_numa1_8
Intel(R) Memory Latency Checker - v3.11a
Command line parameters: --max_bandwidth

Using buffer size of 100.000MiB/thread for reads and an additional 100.000MiB/thread for writes
*** Unable to modify prefetchers (try executing 'modprobe msr')
*** So, enabling random access for latency measurements

Measuring Maximum Memory Bandwidths for the system
Will take several minutes to complete as multiple injection rates will be tried to get the best bandwidth
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using traffic with the following read-write ratios
ALL Reads        :      34667.67
3:1 Reads-Writes :      29677.79
2:1 Reads-Writes :      28391.74
1:1 Reads-Writes :      24642.78
Stream-triad like:      27284.96
```

mem0 7根 共224G mem1 4根 共128G

mlc结果

```
zmq@ubuntu:~/MLC$ cat numa0_7_numa1_3_result
Intel(R) Memory Latency Checker - v3.11a
*** Unable to modify prefetchers (try executing 'modprobe msr')
*** So, enabling random access for latency measurements
Measuring idle latencies for random access (in ns)...
                Numa node
Numa node            0       1
       0          97.1   160.4
       1         160.6    97.2

Measuring Peak Injection Memory Bandwidths for the system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using traffic with the following read-write ratios
ALL Reads        :      51658.0
3:1 Reads-Writes :      49088.5
2:1 Reads-Writes :      48423.2
1:1 Reads-Writes :      46916.7
Stream-triad like:      44409.3

Measuring Memory Bandwidths between nodes within system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using Read-only traffic type
                Numa node
Numa node            0       1
       0        17431.6 29882.5
       1        16373.8 34450.6
```

mem0 7根mem1 4根 同节点和跨节点的最大带宽测试

```
zmq@ubuntu:~/MLC$ cat 00_numa0_7_numa1_4
Intel(R) Memory Latency Checker - v3.11a
Command line parameters: --max_bandwidth

Using buffer size of 100.000MiB/thread for reads and an additional 100.000MiB/thread for writes
*** Unable to modify prefetchers (try executing 'modprobe msr')
*** So, enabling random access for latency measurements

Measuring Maximum Memory Bandwidths for the system
Will take several minutes to complete as multiple injection rates will be tried to get the best bandwidth
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using traffic with the following read-write ratios
ALL Reads        :      17300.56
3:1 Reads-Writes :      14923.17
2:1 Reads-Writes :      14293.95
1:1 Reads-Writes :      12509.10
Stream-triad like:      13899.76

zmq@ubuntu:~/MLC$ cat 01_numa0_7_numa1_4
Intel(R) Memory Latency Checker - v3.11a
Command line parameters: --max_bandwidth

Using buffer size of 100.000MiB/thread for reads and an additional 100.000MiB/thread for writes
*** Unable to modify prefetchers (try executing 'modprobe msr')
*** So, enabling random access for latency measurements

Measuring Maximum Memory Bandwidths for the system
Will take several minutes to complete as multiple injection rates will be tried to get the best bandwidth
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using traffic with the following read-write ratios
ALL Reads        :      34313.00
3:1 Reads-Writes :      29107.20
2:1 Reads-Writes :      27751.97
1:1 Reads-Writes :      22797.71
Stream-triad like:      26697.88
```

带宽分析工具Per-Task Memory Bandwidth Monitoring

https://github.com/intel/PerTaskMemBWMonitoring

三、安装HPC应用openfoam

1. 下载openfoam仓库

```
git clone https://develop.openfoam.com/Development/openfoam.git
```

2. 切换到指定版本

```
cd openfoam
git tag          # 查看所有标签
git checkout tags/OpenFOAM-v2212
```

3. 下载环境依赖，完整说明见doc/Requirements.md · develop · Development / openfoam · GitLab

```
sudo apt-get update
sudo apt-get install build-essential autoconf autotools-dev cmake gawk
gnuplot -y
sudo apt-get install flex libfl-dev libreadline-dev zlib1g-dev openmpi-bin
libopenmpi-dev mpi-default-bin mpi-default-dev -y
sudo apt-get install libgmp-dev libmpfr-dev libmpc-dev -y
sudo apt-get install libfftw3-dev libscotch-dev libptscotch-dev libboost-
system-dev libboost-thread-dev libcgal-dev -y
```

4. 编译，完整说明见doc/Build.md · develop · Development / openfoam · GitLab

```
foamSystemCheck              # 检查系统配置
cd ~/openfoam                # 切换到openfoam目录下
foam                         # 检查环境依赖，没报错就通过
./Allwmake -j -s -q -l       # 编译
```

foamSystemCheck结果：

```
zmq@ubuntu:~/openfoam$ foamSystemCheck

Checking basic system...
--------------------------------------------------------------------------------
Shell:       bash
Host:        ubuntu
OS:          Linux version 4.15.0-213-generic

System check: PASS
==================
Can continue to OpenFOAM installation.
```

5. 测试

```
source ~/openfoam/etc/bashrc
foamInstallationTest         # 验证build是否正确
foamTestTutorial -full incompressible/simpleFoam/pitzDaily      # 测试指定案例
```

foamInstallationTest结果：

```
Summary
--------------------------------------------------------------------------------
Base configuration ok.
Critical systems ok.

Done
```

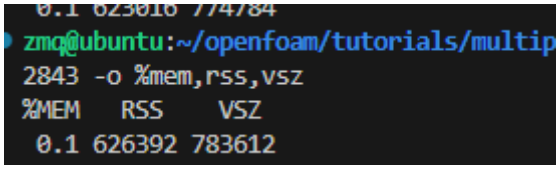
foamTestTutorial结果：

```
zmq@ubuntu:~/openfoam/etc$ foamTestTutorial -full incompressible/simpleFoam/pitzDaily
Run test: incompressible/simpleFoam/pitzDaily
Running blockMesh on /tmp/tmp.97BB7sKXwx.incompressible_simpleFoam_pitzDaily
Running simpleFoam on /tmp/tmp.97BB7sKXwx.incompressible_simpleFoam_pitzDaily
run: OK
Passed all 1 tests
```

6. 实例

   实例在`tutorials`文件夹下，找到需要执行的示例通过`./Allrun`执行。

   测试`/openfoam/tutorials/multiphase/interPhaseChangeFoam/cavitatingBullet`



[OpenFOAM笔记 | 一、介绍_openfoam中如果想知道一个命令怎么用,输入help-CSDN博客]

四、安装HPL

1. 下载HPL 2.1 源码

```
wget http://www.netlib.org/benchmark/hpl/hpl-2.1.tar.gz
tar -zxvf hpl-2.1.tar.gz
mv hpl-2.1 hpl      # 修改文件名，否则编译的时候路径不对会报错
```

2. 编译

   执行`setup`目录下的`make_generic`脚本，生成`Make.UNKNOWN`文件

```
cd setup
chmod +x make_generic
./make_generic
cd ../
cp setup/Make.UNKNOWN Make.Linux
# 修改 Make.Linux 和 Make.top 中的 ARCH = UNKNOWN 为
ARCH         = Linux
# 安装 blas 库
sudo apt-get install libblas-dev libopenblas-dev
make arch=Linux
```

3. 执行

```
cd bin/Linux/
~/pintool/pin -t ~/pintool/source/tools/ManualExamples/obj-
intel64/pinatrace.so -- mpirun -np 4 xhpl
```

4. 修改`HPL.dat`参数测试更多示例，参数说明见`TUNING`，主要修改5678行

```
# Line 5 (问题规模数量): 指定要执行的问题规模数量，必须小于等于 20
3
# Line 6 (问题规模列表): 指定要运行的问题规模
300 600 1000 NS
# Line 7 (块大小数量): 指定要运行的块大小数量，必须小于等于 20
4
# Line 8 (块大小列表): 指定要使用的块大小
80 100 120 140
```

五、安装HPCG

1. 下载源码

```
git clone https://github.com/hpcg-benchmark/hpcg.git
```

2. 编译

```
# 安装 openmpi
sudo apt install openmpi-bin openmpi-common libopenmpi-dev
# 查看 openmpi 是否配置正确
mpicc --version
mpirun --version
# 安装 gcc 11 版本 并 切换
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo apt update
sudo apt install gcc-11 g++-11
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-11 100
sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-11 100
sudo update-alternatives --config gcc
sudo update-alternatives --config g++
# 创建 build 目录
cd hpcg
mkdir build
cd build
../configure Linux_MPI
make
```

3. 测试

```
cd build/bin
# 设置线程数为 8
export OMP_NUM_THREADS=8
# 修改pinatrace.cpp中的输出文件为pinatrace_pid.out，将不同进程的输出到不同文件中
mpirun -np 16 ~/pintool/pin -t ~/pintool/source/tools/ManualExamples/obj-intel64/pinatrace.so -- xhpcg
```

4. 修改`hpcg.dat`参数测试更多示例，参数说明见`TUNING`，设太大会爆内存

```
# line 3 指定每个 MPI 进程的局部问题维度
104 104 104
# line 4 指定计时部分的基准测试运行的时间（秒）
60
```

六、安装WRF

1. 下载源码

```
git clone https://github.com/wrf-model/WRF.git
```

2. 切换到指定版本

```
cd WRF
git tag             # 查看所有标签
git checkout tags/v4.2
```

3. 配置环境

```
cd ~
mkdir Build_WRF
cd Build_WRF
mkdir LIBRARIES
cd LIBRARIES
# 下载库文件
wget
https://www2.mmm.ucar.edu/wrf/OnLineTutorial/compile_tutorial/tar_files/mpic
h-3.0.4.tar.gz
wget
https://www2.mmm.ucar.edu/wrf/OnLineTutorial/compile_tutorial/tar_files/netc
df-c-4.7.2.tar.gz
wget
https://www2.mmm.ucar.edu/wrf/OnLineTutorial/compile_tutorial/tar_files/netc
df-fortran-4.5.2.tar.gz
wget
https://www2.mmm.ucar.edu/wrf/OnLineTutorial/compile_tutorial/tar_files/jasp
er-1.900.1.tar.gz
wget
https://www2.mmm.ucar.edu/wrf/OnLineTutorial/compile_tutorial/tar_files/libp
ng-1.2.50.tar.gz
wget
https://www2.mmm.ucar.edu/wrf/OnLineTutorial/compile_tutorial/tar_files/zlib
-1.2.11.tar.gz
# 设置环境变量，在.bashrc中添加以下内容
export DIR=~/Build_WRF/LIBRARIES
export CC=gcc
export CXX=g++
export FC=gfortran
export FCFLAGS=-m64
export F77=gfortran
export FFLAGS=-m64
export JASPERLIB=$DIR/grib2/lib
export JASPERINC=$DIR/grib2/include
export LDFLAGS=-L$DIR/grib2/lib
export CPPFLAGS=-I$DIR/grib2/include
# 使环境变量生效
source .bashrc
# 安装 NetCDF-C
tar -zxvf netcdf-c-4.7.2.tar.gz
```

```
cd netcdf-c-4.7.2/
sudo CFLAGS="-fPIC" ./configure --prefix=$DIR/netcdf --disable-dap --
disable-netcdf-4 --disable-shared
make
sudo make install
# 设置环境变量
export PATH=$DIR/netcdf/bin:$PATH
export NETCDF=$DIR/netcdf
cd ..
export LIBS="-lnetcdf -lz"
# 安装 NetCDF-fortran
tar -zxvf netcdf-fortran-4.5.2.tar.gz
cd netcdf-fortran-4.5.2/
./configure --prefix=$DIR/netcdf CPPFLAGS=-I$DIR/netcdf/include LDFLAGS=-
L$DIR/netcdf/lib
make
sudo make install
export PATH=$DIR/netcdf/bin:$PATH
export LD_LIBRARY_PATH=${NETCDF}/lib:$LD_LIBRARY_PATH
cd ..
# 安装 MPICH， to run WRF with multiple processors
tar -zxvf mpich-3.0.4.tar.gz
cd mpich-3.0.4/
sudo apt-get install libnetcdf-dev
# 切换gcc版本到10 或 更低
sudo update-alternatives --config gcc
./configure --prefix=$DIR/mpich
make
sudo make install
export PATH=$DIR/mpich/bin:$PATH
cd ..
# 安装 zlib
tar -zxvf zlib-1.2.11.tar.gz
cd  zlib-1.2.11/
./configure --prefix=$DIR/grib2
make
make install
cd ..
# 安装libpng
tar -zxvf libpng-1.2.50.tar.gz
cd libpng-1.2.50/
./configure --prefix=$DIR/grib2
make
make install
cd ..
# 安装Jasper
tar -zxvf jasper-1.900.1.tar.gz
cd jasper-1.900.1/
./configure --prefix=$DIR/grib2
make
make install
cd ..
# 测试 Fortran + C + NetCDF
cd ~
mkdir TESTS
```

```
cd TESTS
wget
https://www2.mmm.ucar.edu/wrf/OnLineTutorial/compile_tutorial/tar_files/Fort
ran_C_NETCDF_MPI_tests.tar
tar -xf Fortran_C_NETCDF_MPI_tests.tar
cp ${NETCDF}/include/netcdf.inc .
# 编译Fortran源文件
gfortran -c 01_fortran+c+netcdf_f.f
# 编译C源文件
gcc -c 01_fortran+c+netcdf_c.c
# 链接
gfortran 01_fortran+c+netcdf_f.o 01_fortran+c+netcdf_c.o -L${NETCDF}/lib -
lnetcdff -lnetcdf
# 执行
./a.out
# 测试 Fortran + C + NetCDF + MPI
cp ${NETCDF}/include/netcdf.inc .
mpif90 -c 02_fortran+c+netcdf+mpi_f.f
mpicc -c 02_fortran+c+netcdf+mpi_c.c
mpif90 02_fortran+c+netcdf+mpi_f.o 02_fortran+c+netcdf+mpi_c.o -
L${NETCDF}/lib -lnetcdff -lnetcdf
mpirun ./a.out
```

4. 检查环境无误后，编译WRF

```
cd WRF
./configure     #选33
# 下载 C Shell解释器
sudo apt-get install csh
sudo ln -s /usr/bin/cpp /lib/cpp
./compile em_grav2d_x
```

5. 运行

```
cd run
~/pintool/pin -t ~/pintool/source/tools/ManualExamples/obj-
intel64/pinatrace.so -- ./ideal.exe
```

6.

   官方文档：https://www2.mmm.ucar.edu/wrf/OnLineTutorial/compilation_tutorial.php

   configure: error: netcdf.h could not be found. Please set CPPFLAGS：https://blog.csdn.net/weixin_51083023/article/details/136819596

七、安装openblas

1. 下载源码

```
git clone https://github.com/OpenMathLib/OpenBLAS.git
```

2. 编译

```
cd OpenBLAS
# 切换 gfortran 版本和 gcc 一致
make
sudo make install # 默认安装到 /opt/OpenBLAS 目录下
```

3. 运行

```
# 代码在 USAGE.md 中
gcc -o test_cblas_open test_cblas_dgemm.c -I /opt/OpenBLAS/include/ -
L/opt/OpenBLAS/lib -lopenblas -lpthread -lgfortran
~/pintool/pin -t ~/pintool/source/tools/ManualExamples/obj-
intel64/pinatrace.so -- ./test_cblas_open
gcc -o time_dgemm time_dgemm.c /opt/OpenBLAS/lib/libopenblas.a -lpthread
~/pintool/pin -t ~/pintool/source/tools/ManualExamples/obj-
intel64/pinatrace.so -- ./time_dgemm 1000 1000 1000
```

八、使用Perf分析内存

1. 安装Perf

```
# 下载linux-tools-common
sudo apt-get install linux-tools-common linux-tools-generic linux-
tools-$(uname -r)
# 查看是否存在perf
perf --version
```

2. 使用perf分析应用的cache miss次数和中断次数

```
perf stat -e cache-misses,mem-loads,mem-stores,hw_interrupts.received
<command>
```

```
# 临时设置perf权限
cat /proc/sys/kernel/perf_event_paranoid    # 查看当前设置 3
sudo sysctl -w kernel.perf_event_paranoid=-1 # 设置为-1，允许几乎所有事件
cat /proc/sys/kernel/kptr_restrict      # 默认为1
sudo sysctl -w kernel.kptr_restrict=0   # 设置为0
# 查看当前perf的最大采样频率
cat /proc/sys/kernel/perf_event_max_sample_rate
# 修改当前perf的最大采样频率
sudo sysctl -w kernel.perf_event_max_sample_rate=31750
# 使用perf record，-F 设置采样频率
perf record -e cache-misses,mem-loads,mem-stores,hw_interrupts.received -F
1000 <command>
perf record -e cache-misses,mem-loads,mem-stores,hw_interrupts.received -F
1000 -p <pid>
perf report
```

```
perf mem record -F 1000 <command>
perf mem report
# 处理 perf 结果
perf script -i perf.data > output.txt

perf record -F 10000 -e cpu/mem-loads,ldlat=30/P -e cpu/mem-stores/P -d -o
perf.data -- ./your_program
perf script -i perf.data

perf record -F 10000 -e cpu/mem-loads,ldlat=30/P -e cpu/mem-stores/P -e
l2_rqsts.miss -e l2_rqsts.all_demand_miss -e l2_rqsts.code_rd_miss -e
l2_rqsts.demand_data_rd_miss -e l2_rqsts.rfo_miss -e l2_rqsts.pf_miss -d -o
perf.data ./your_program
```

WARNING: perf not found for kernel 5.15.19解决办法：

```
# 删除/usr/bin/perf
sudo rm /usr/bin/perf
# 创建软链接
sudo ln -s /usr/lib/linux-tools/5.4.0-192-generic/perf /usr/bin/perf
```

https://blog.csdn.net/qq_36573282/article/details/122181866

九、使用pintool获取内存访问序列

1. 安装pintool

```
wget https://software.intel.com/sites/landingpage/pintool/downloads/pin-
3.30-98830-g1d7b601b3-gcc-linux.tar.gz
tar -xzf pin-3.30-98830-g1d7b601b3-gcc-linux.tar.gz
mv pin-3.30-98830-g1d7b601b3-gcc-linux pintool
rm pin-3.30-98830-g1d7b601b3-gcc-linux.tar.gz
```

2. 修改pinatrace.cpp

```
# copy pinatrace.cpp
cp pinatrace.cpp pintool/source/tools/ManualExamples/
```

3. 编译

```
cd pintool/source/tools/ManualExamples/
make obj-intel64/pinatrace.so TARGET=intel64
```

4. 修改 `/openfoam/bin/tools/RunFunctions` 加入 pintool 追踪内存访问

```
# 修改 runApplication() 中的
$appRun $appArgs "$@" >> $logFile 2>&1
# ↑ 修改为 ↓
~/pintool/pin -t ~/pintool/source/tools/ManualExamples/obj-
intel64/pinatrace.so -- $appRun $appArgs "$@" >> $logFile 2>&1
# 修改文件名，避免覆盖
mv pinatrace.out pinatrace."$appName"
```

```
$appRun $appArgs "$@" > $logFile 2>&1
# ↑ 修改为 ↓
~/pintool/pin -t ~/pintool/source/tools/ManualExamples/obj-
intel64/pinatrace.so -- $appRun $appArgs "$@" > $logFile 2>&1
# 修改文件名，避免覆盖
mv pinatrace.out pinatrace."$appName"

# 修改 runParallel() 中的
$mpirun -n $nProcs $appRun $appArgs "$@" </dev/null >> $logFile 2>&1
# ↑ 修改为 ↓
~/pintool/pin -t ~/pintool/source/tools/ManualExamples/obj-
intel64/pinatrace.so -- $mpirun -n $nProcs $appRun $appArgs "$@" </dev/null
>> $logFile 2>&1
# 修改文件名，避免覆盖
mv pinatrace.out pinatrace."$appName"

$mpirun -n $nProcs $appRun $appArgs "$@" </dev/null > $logFile 2>&1
# ↑ 修改为 ↓
~/pintool/pin -t ~/pintool/source/tools/ManualExamples/obj-
intel64/pinatrace.so -- $mpirun -n $nProcs $appRun $appArgs "$@" </dev/null
> $logFile 2>&1
# 修改文件名，避免覆盖
mv pinatrace.out pinatrace."$appName"
```

5. 复制应用示例到`openfoam/tracetutorials`目录下并执行

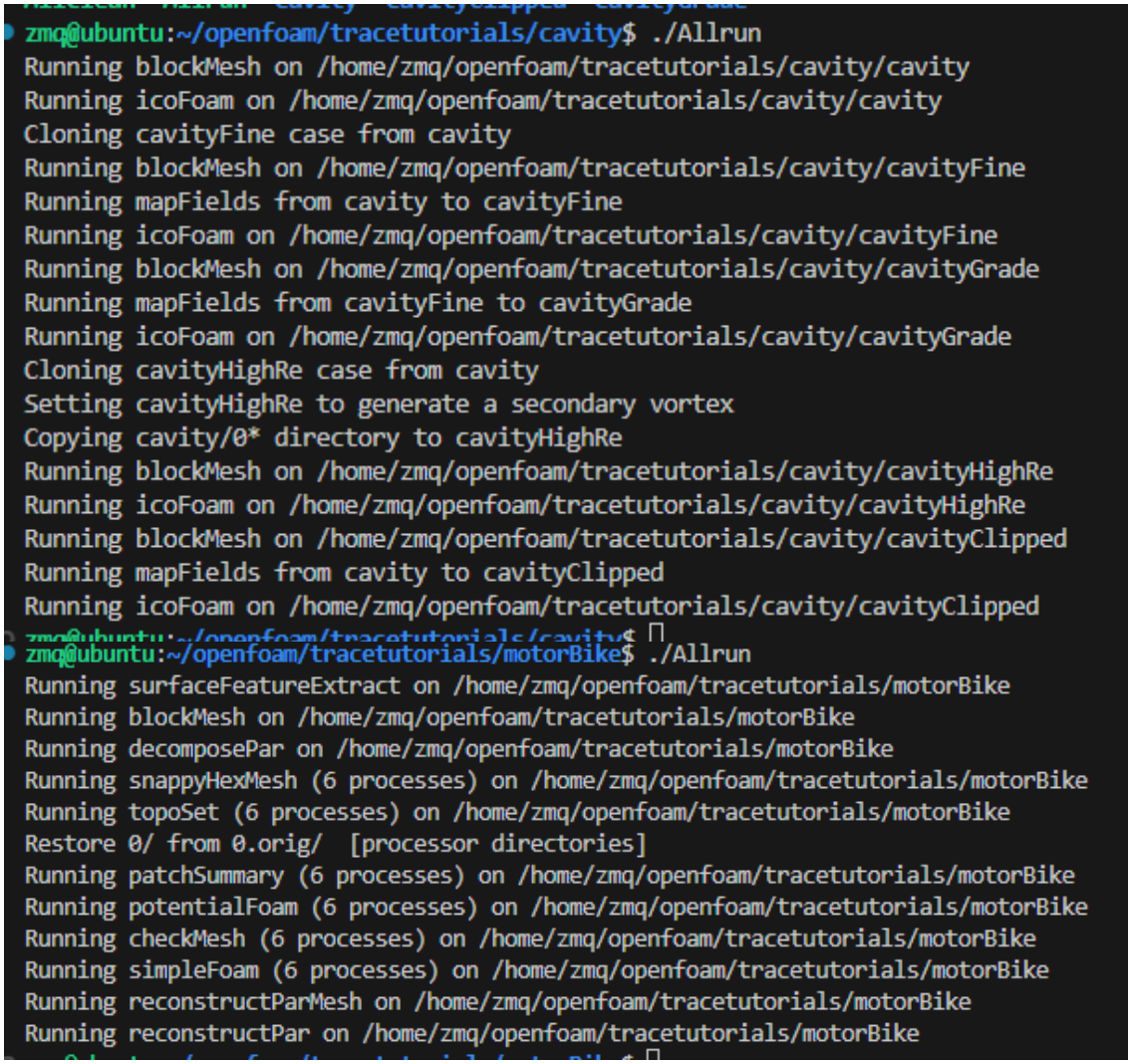
```
cd openfoam
mkdir tracetutorials
cp -r /tutorials/multiphase/interPhaseChangeFoam/cavitatingBullet
tracetutorials
cd tracetutorials/cavitatingBullet
./Allrun
```

6. 模拟L3-cache过滤访问序列

```
# 查看系统L3-cache大小
lscpu
# 查看每层cache的策略
sudo dmidecode -t cache
# 修改L3-Cache.cpp中的缓存设置与系统一致

    #define CACHELINE 0x3f
    #define L1_SET 64
    #define L1_WAY 8
    #define L2_SET 1024
    #define L2_WAY 16
    #define L3_SET 225280
    #define L3_WAY 1

# 编译
g++ -g -o l3-cache L3-Cache.cpp
# 执行
./l3-cache pinatrace cache_miss
```



十、ARM架构上安装使用SPE

1. 安装perf

```
sudo dnf install perf
```

2. 在BIOS中启用SPE，并在linux启动项里设置 kpti=off

```
BIOS  Advanced -> MISC Config -> SPE Enable
vim /etc/grub2-efi.cfg
找到对应内核版本的开机启动项，在末尾增加 kpti=off  关闭Kernel Page Table Isolation（内核页表隔离）
```

   https://www.hikunpeng.com/forum/thread-0223142082297176038-1-1.html

3. perf list | grep arm_spe查看是否开启成功
4. 使用SPE分析应用

```
perf record -e arm_spe// -c 1000000 -- ./mybench

# SPE允许通过配置参数对采样数据进行过滤，以下是常见的配置参数：
# branch_filter=1：仅采集分支指令的样本。
# load_filter=1：仅采集内存加载操作。
# store_filter=1：仅采集内存存储操作。
# 采样
perf record -e arm_spe/store_filter=1/ -- ./mybench
# 分析
perf report
perf report --mem-mode      # 指令的内存访问详情
perf report --itrace=i1i    # 生成指令级别的样本
```

https://perf.wiki.kernel.org/index.php/Latest_Manual_Page_of_perf-arm-spe.1

十一、复现Memtis

1. 下载源代码

```
git clone https://github.com/cosmoss-jigu/memtis.git
```

2. 安装linux 内核

查看当前内核版本

```
zmq@pm110:~/memtis/linux$ uname -a
Linux pm110 5.4.0-190-generic #210-Ubuntu SMP Fri Jul 5 17:03:38 UTC 2024 x86_64 x86_64 x86_64 GNU/Linux
```

安装工具包

```
sudo apt-get install libncurses5-dev libssl-dev
sudo apt-get install build-essential openssl
sudo apt-get install zlibc minizip
sudo apt-get install libidn11-dev libidn11
sudo apt install bison
sudo apt install libssl-dev
sudo apt-get install dwarves
sudo apt-get install flex
```

```
[ ] Read-only THP for filesystems (EXPERIMENTAL)
[*] Enable hugepage-aware tiered memory management
    Data Access Monitoring
<*>   Intel Uncore frequency control driver
```

```
cd memtis/linux
sudo make mrproper  # 清除编译以来产生的所有中间文件
sudo make clean     # 清除上次编译产生的中间文件
sudo make menuconfig        # 生成图形化界面
# 设置 CONFIG_HTMM=y
# make[1]: *** No rule to make target 'debian/canonical-certs.pem', needed
by 'certs/x509_certificate_list'.  Stop.
# 清空 .config文件中这两行，需要在BIOS里关闭secure boot
CONFIG_SYSTEM_TRUSTED_KEYS=""
CONFIG_SYSTEM_REVOCATION_KEYS=""
```

```
# 编译
sudo make -j 8
# 安装内核和模块
sudo make modules_install
sudo make install
sudo update-grub
sudo reboot # 重启
# 出现报错
# Loading initial ramdis ...
# error: out of memory
# Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)
sudo cp /etc/initramfs-tools/initramfs.conf /etc/initramfs-tools/conf.d/
# 修改conf.d中的conf文件 MODULES=dep
# 删掉 /boot/目录下的 initrd.img-5.15.19-htmm
sudo rm /boot/initrd.img-5.15.19-htmm
sudo update-initramfs -c -k 5.15.19-htmm
sudo update-grub
```

参考链接
https://unix.stackexchange.com/questions/698890/out-of-memory-on-loading-initial-ramdisk-after-kernel-upgrade-4-15-to-4-19-o
https://unix.stackexchange.com/questions/270390/how-to-reduce-the-size-of-the-initrd-when-compiling-your-kernel
https://stackoverflow.com/questions/67670169/compiling-kernel-gives-error-no-rule-to-make-target-debian-certs-debian-uefi-ce

```
# 启用 cgroup v2
# 编辑/etc/default/grub 文件，在 GRUB_CMDLINE_LINUX 行，移除cgroup.memory=nokmem 参数，添加 systemd.unified_cgroup_hierarchy=1 参数。
sudo update-grub
sudo reboot
# 查看 cgroup v2 是否启用
mount | grep cgroup2
```

3. 运行

```
cd /memtis/memtis-userspace
make
./scripts/run_bench.sh -B ${BENCH} -R ${NR} -V testP -NS -NW --cxl
```

4. 构建benchmark silo

```
# 构建 masstree
cd /bench_dir/silo/masstree
./configure
make -j8
./mttest    # 基本测试
# 构建 silo
cd silo
sudo apt-get install libdb++-dev
sudo apt-get install libaio-dev
MODE=perf make -j dbtest
```

5. 修改脚本 run_bench.sh 中绑定cpu的行

```
PINNING="taskset -c $(seq -s, 0 2 38)"
```

```
/usr/bin/time -v ./your_application      #获取应用的最大内存占用
# 查看日志
journalctl --since "2023-10-01 08:00:00" --until "2023-10-01 10:00:00"
awk '$0 >= "Oct 17 06:00:00" && $0 <= "Oct 17 07:00:00"' /var/log/syslog
```

[8.289194] ACPI Error: No handler for Region [SYSI] (000000000163e2fb) [IPMI] (20210730/evregion-130)
[8.290320] ACPI Error: Region IPMI (ID=7) has no handler (20210730/exfldio-261)

[8.291540] No Local Variables are initialized for Method [_GHL]

[8.291542] No Arguments are initialized for method [_GHL]

[8.291547] ACPI Error: Aborting method _SB.PMI0.*GHL due to previous error (AE_NOT_EXIST) (20210730/psparse-529)*
*[8.292832] ACPI Error: Aborting method _SB.PMI0.*PMC due to previous error (AE_NOT_EXIST) (20210730/psparse-529)

systemd-networkd.service: Job network-pre.target/start deleted to break ordering cycle starting with systemd-networkd.service/start

Memtis代码解析

1. htmm_sampler.c中包含采样相关函数

   `ksamplingd_init`首先初始化pebs，然后启动采样线程

   `pebs_init`首先为mem_event分配内存空间，然后为通过`__perf_event_open`为每个事件设置采样信息，通过`htmm__perf_event_init`初始化环形缓冲区

   在node0的一个CPU中运行采样进程，获取所有CPU的监测事件。`ksamplingd`为采样的主体函数，获取采样信息并统计CPU开销。在主体for循环中读取环形缓冲区获取采样到的`pid`、`addr`和`event`，调用`update_pginfo`更新页面信息，并统计事件数，计算cpu使用率，通过调用`pebs_update_period`更新采样频率。正常情况下每隔2毫秒读取一次环形缓冲区。

   `update_pginfo`函数在htmm_core.c中

2. htmm_migrater.c中包含页面迁移相关函数

   `kmigraterd_init`为系统中的每个内存节点创建和初始化一个内存迁移线程，在`kmigraterd`中如果是htmm_cxl_mode，则对node0执行降级，对node1执行提升（修改点），迁移线程在页面迁出的节点上运行。**页面分裂是什么?**

```
//    if (htmm_cxl_mode) {
//  if (nid == 0)
//      return kmigraterd_demotion(pgdat);
//  else
//      return kmigraterd_promotion(pgdat);
//    }
    if (htmm_cxl_mode) {
    if (nid == 0)
        return kmigraterd_promotion(pgdat);
    else
        return kmigraterd_demotion(pgdat);
    }
```

`demote_node`和`promote_node`实现页面的迁移，修改其中在htmm_cxl_mode下的目标节点

`demote_lruvec`和`promote_lruvec`将页面从当前节点迁移至目标节点

`demote_inactive_list`和`promote_active_list`实际执行页面迁移操作

`migrate_page_list`根据是页面提升还是降级选择目标节点`nid``（修改点）`，调用`migrate_pages`执行页面迁移

3. htmm_core.c中包含页面信息更新相关函数

   `__update_pmd_pginfo`中调用`update_huge_page`更新大页信息，并根据当前节点是否是顶层节点返回状态值`（修改点）`；`__update_pte_pginfo`与之类似，调用`update_base_page`更新页面信息，并根据当前节点是否是顶层节点返回状态值`（修改点）`

内存管理笔记

1. 进程无论是在用户态还是在内核态能够看到的都是虚拟内存空间，物理内存空间被操作系统所屏蔽进程是看不到的。进程通过虚拟内存地址访问这些数据结构的时候，虚拟内存地址会在内存管理子系统中被转换成物理内存地址，通过物理内存地址就可以访问到真正存储这些数据结构的物理内存了。
2. 结构体`vm_area_struct`描述了这些虚拟内存区域 VMA（virtual memory area）,vm_page_prot 和 vm_flags 都是用来标记 vm_area_struct 结构表示的这块虚拟内存区域的访问权限和行为规范。`vm_page_prot`偏向于定义底层内存管理架构中页这一级别的访问控制权限，它可以直接应用在底层页表中，它是一个具体的概念。`vm_flags`则偏向于定于整个虚拟内存区域的访问权限以及行为规范。`vm_flags`的值中`VM_SEQ_READ`的设置用来暗示内核，应用程序对这块虚拟内存区域的读取是会采用顺序读的方式进行，内核会根据实际情况决定预读后续的内存页数，以便加快下次顺序访问速度。`VM_RAND_READ`的设置会暗示内核，应用程序会对这块虚拟内存区域进行随机读取，内核则会根据实际情况减少预读的内存页数甚至停止预读。我们可以通过`posix_fadvise`，`madvise`系统调用来暗示内核是否对相关内存区域进行顺序读取或者随机读取。
3. 在内核中其实是通过一个 struct vm_area_struct 结构的双向链表将虚拟内存空间中的这些虚拟内存区域 VMA 串联起来的。vm_area_struct 结构中的 vm_next，vm_prev 指针分别指向 VMA 节点所在双向链表中的后继节点和前驱节点，内核中的这个 VMA 双向链表是有顺序的，所有 VMA 节点按照低地址到高地址的增长方向排序。在内核中，同样的内存区域 vm_area_struct 会有两种组织形式，一种是双向链表用于高效的遍历，另一种就是红黑树用于高效的查找。
4. 内核态虚拟内存空间是所有进程共享的，不同进程进入内核态之后看到的虚拟内存空间全部是一样的。进程进入内核态之后使用的仍然是虚拟内存地址，只不过在内核中使用的虚拟内存地址被限制在了内核态虚拟内存空间范围中。

5. FLATMEM 平坦内存模型只适合管理一整块连续的物理内存，而对于多块非连续的物理内存来说使用 FLATMEM 平坦内存模型进行管理则会造成很大的内存空间浪费。因为 FLATMEM 平坦内存模型是利用 mem_map 这样一个全局数组来组织这些被划分出来的物理页 page 的，而对于物理内存存在大量不连续的内存地址区间这种情况时，这些不连续的内存地址区间就形成了内存空洞。
6. 在 DISCONTIGMEM 非连续内存模型中，内核将物理内存从宏观上划分成了一个一个的节点 node（微观上还是一页一页的物理页），每个 node 节点管理一块连续的物理内存。这样一来这些连续的物理内存页均被划归到了对应的 node 节点中管理，就避免了内存空洞造成的空间浪费。
7. SPARSEMEM 稀疏内存模型的核心思想就是对粒度更小的连续内存块进行精细的管理，用于管理连续内存块的单元被称作 section 。在 struct page 结构中有一个 `unsigned long flags` 属性，在 flag 的高位 bit 中存储着 page 所在 mem_section 数组中的索引，从而可以定位到所属 section。PFN 的高位 bit 存储的是全局数组 mem_section 中的 section 索引，PFN 的低位 bit 存储的是 section_mem_map 数组中具体物理页 page 的索引。
8. UMA 架构的优点很明显就是结构简单，所有的 CPU 访问内存速度都是一致的，都必须经过总线。然而它的缺点我刚刚也提到了，就是随着处理器核数的增多，总线的带宽压力会越来越大。解决办法就只能扩宽总线，然而成本十分高昂，未来可能仍然面临带宽压力。为了解决以上问题，提高 CPU 访问内存的性能和扩展性，于是引入了一种新的架构：非一致性内存访问 NUMA（Non-uniform memory access）。在 NUMA 架构下，只有 DISCONTIGMEM 非连续内存模型和 SPARSEMEM 稀疏内存模型是可用的。
9. 可以在应用程序中通过 libnuma 共享库中的 API 调用 set_mempolicy 接口设置进程的内存分配策略。
10. 系统中所有 NUMA 节点中的物理页都是依次编号的，每个物理页的 PFN 都是**全局唯一的**（不只是其所在 NUMA 节点内唯一）。
11. 内核通过迁移页面来规整内存，这样就可以避免内存碎片，从而得到一大片连续的物理内存，以满足内核对大块连续内存分配的请求。所以这就是内核需要根据物理页面是否能够迁移的特性，而划分出 ZONE_MOVABLE 区域的目的。
12. **内核会为每个 NUMA 节点分配一个 kswapd 进程用于回收不经常使用的页面，还会为每个 NUMA 节点分配一个 kcompactd 进程用于内存的规整避免内存碎片。**

```
typedef struct pglist_data {
        .........
    // 页面回收进程
    struct task_struct *kswapd;
    wait_queue_head_t kswapd_wait;
    // 内存规整进程
    struct task_struct *kcompactd;
    wait_queue_head_t kcompactd_wait;

        ..........
} pg_data_t;
```

内存页面迁移相关

`migrate_pages()` 系统调用是页面迁移的主要接口

MR_SYSCALL	**应用层主动调用migrate_pages()或move_pages()触发的迁移。**
MR_MEMPOLICY_MBIND	调用mbind系统调用设置memory policy时触发的迁移
MR_NUMA_MISPLACED	numa balance触发的页面迁移

页面迁移不是简单的把一个page从A位置移动到B位置， 它的本质是一个分配新页面， 将旧页面的内容拷贝至新页面， 解除旧页面的映射关系，并将映射关系映射到新页面，最后释放旧页面的过程。

`mbind`是一个Linux系统调用函数，它允许在进程地址空间内把一段连续的内存页与特定的NUMA节点绑定。这可以确保相关的数据和程序状态存储在 NUMA节点上，从而提高内存访问性能。

禁用numa balance

```
# 查看 numa balance 是否开启，0为关闭，1为开启
cat /proc/sys/kernel/numa_balancing
# 动态关闭 NUMA balance ，重启后失效
echo 0 | sudo tee /proc/sys/kernel/numa_balancing
```

```
# 查看指定进程的内存使用 rss为实际使用的物理内存 vsz为虚拟内存
ps -p <PID> -o %mem,rss,vsz
# pintool 监测指定应用程序
~/pintool/pin -t ~/pintool/source/tools/ManualExamples/obj-intel64/pinatrace.so
-- ./Allrun
# 查找包含指定内容的行
grep -n 'line' file
# 查看文件指定行
sed -n '100p' file.txt
# 查看进程的虚拟内存空间布局以及其中包含的所有内存区域
pmap pid
# 查看具体物理内存布局情况
cat /proc/iomem
# 查看系统中各个 NUMA 节点中的各个内存区域的内存使用情况
cat /proc/zoneinfo
```

```
# 调试 pintool
# 编译时启动 DEBUG 选项
make DEBUG=1 obj-intel64/pinatrace.so TARGET=intel64
# 执行时增加 --appdebug
/home/zmq/pintool/pin -appdebug -t
/home/zmq/pintool/source/tools/ManualExamples/obj-intel64/pinatrace.so --
./hello_world
# gdb 打开指定文件
gdb obj-intel64/pinatrace.so
# 绑定到远程进程
target remote :40159
# 设置断点，进行调试
```

```
# 挂载磁盘
sudo mount /dev/sda3 /home/zmq/traceout
# 取消挂载
sudo umount /home/zmq/traceout
```

禁用和启用L3 cache

Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A 11.5.3-11.5.4

https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf

启动时系统盘分区丢失

1. 在另一个盘中重装系统，进去后安装TestDisk，并进入

```
sudo apt install testdisk
sudo testdisk
```

2. 使用testdisk恢复分区信息

```
选择“Create”后，按 Enter。
然后选择您要分析的磁盘（在此情况下是 /dev/nvme0n1），并确认。
选择“EFI GPT”作为分区类型（通常适用于大多数情况）。
接下来，选择“Analyse”进行分区分析。
选择快速搜索（Quick Search），TestDisk 将扫描已知的分区。
如果确认分区和文件都正常，您可以选择将这些分区写入磁盘。在 TestDisk 界面中，选择
“Write”（写入）并确认写入操作。
lsblk   # 查看磁盘分区是否恢复
sudo lvdisplay  # 查看能否识别到逻辑卷
sudo reboot     # 重启，系统将恢复
```

系统自动休眠

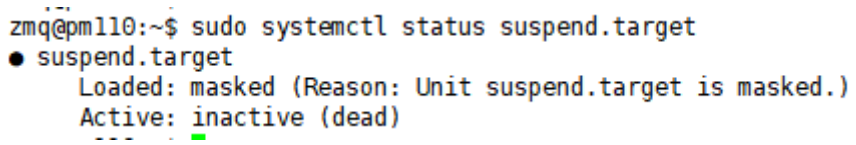
日志信息

```
Oct 20 10:04:04 pm110 NetworkManager[1877]: <info>  [1729418644.0541] manager:
sleep: sleep requested (sleeping: no  enabled: yes)
Oct 20 10:04:04 pm110 NetworkManager[1877]: <info>  [1729418644.0543] manager:
NetworkManager state is now ASLEEP
Oct 20 10:04:04 pm110 ModemManager[1979]: <info>  [sleep-monitor] system is
about to suspend
Oct 20 10:04:04 pm110 gnome-shell[2282]: Screen lock is locked down, not locking
Oct 20 10:04:04 pm110 systemd[1]: Reached target Sleep.
Oct 20 10:04:04 pm110 systemd[1]: Starting Record successful boot for GRUB...
Oct 20 10:04:04 pm110 systemd[1]: Starting Suspend...
```

解决方法

```
zmq@pm110:~$ sudo systemctl status suspend.target
● suspend.target - Suspend
     Loaded: loaded (/lib/systemd/system/suspend.target; static; vendor preset: enabled)
     Active: inactive (dead)
       Docs: man:systemd.special(7)
zmq@pm110:~$ sudo systemctl mask sleep.target suspend.target hibernate.target hybrid-sleep.target
Created symlink /etc/systemd/system/sleep.target → /dev/null.
Created symlink /etc/systemd/system/suspend.target → /dev/null.
Created symlink /etc/systemd/system/hibernate.target → /dev/null.
Created symlink /etc/systemd/system/hybrid-sleep.target → /dev/null.
```

```
# 禁用休眠
systemctl mask sleep.target suspend.target hibernate.target hybrid-sleep.target
# 恢复休眠
systemctl unmask sleep.target suspend.target hibernate.target hybrid-
sleep.target
```



参考链接:

https://askubuntu.com/questions/1271282/how-to-stop-networkmanager-to-put-ubuntu-server-to-sleep