探究OpenGauss数据库的性能

第二小组 游克垚 123037910062 高鹏轩 123037910032 李天策 123037910063 洪墅霖 123037910028

实验环境

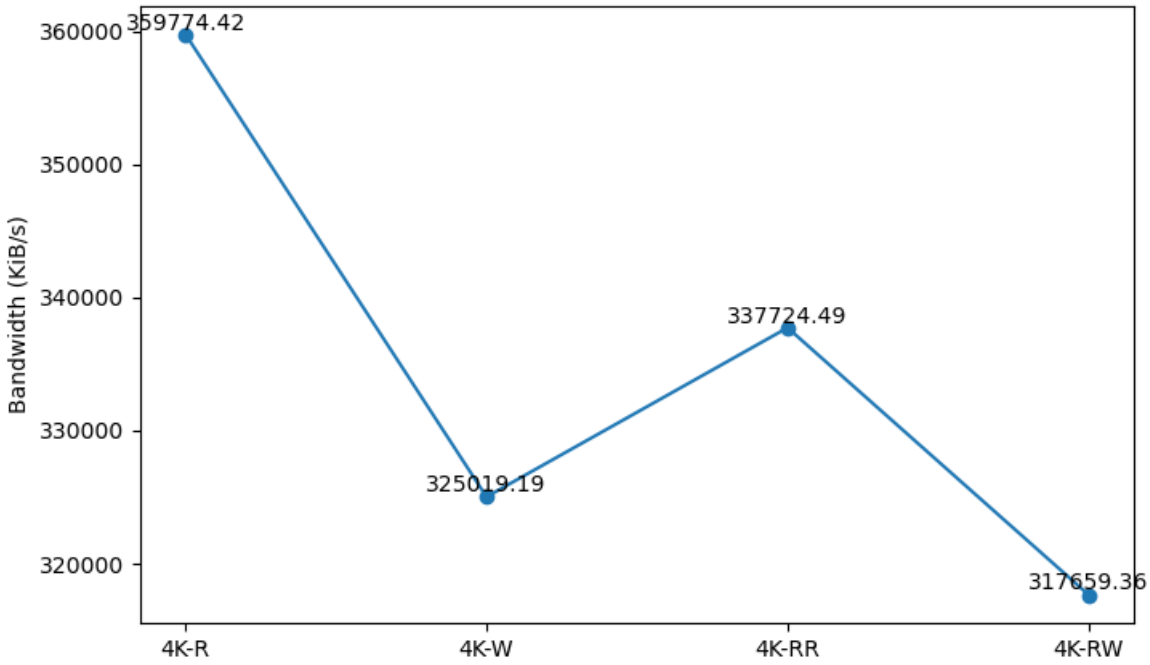
- **硬件设备**
  - **CPU**：Kunpeng-920(aarch64) 128cores 128threads,内存:256GB
  - **硬盘**：256G
  - **系统**：openEuler 22.03
  - **fio**: 3.27
- **软件依赖**

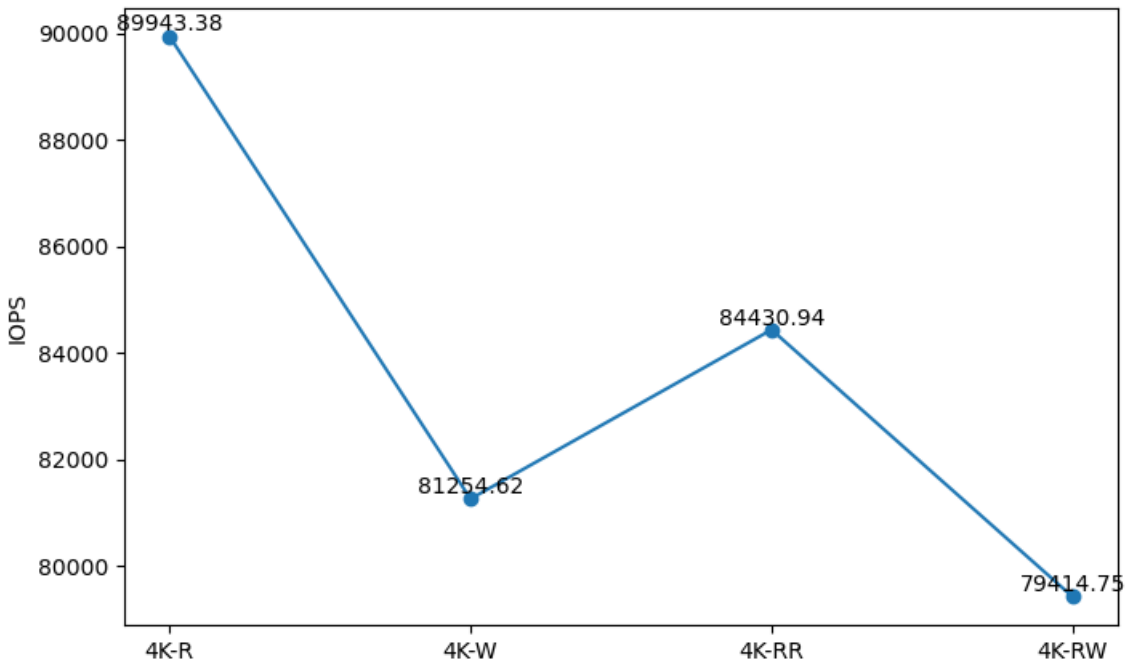
软件	推荐版本	下载版本
libaio-devel	0.3.109-13	0.3.112-2
flex	2.5.31 以上	2.6.4-3
bison	2.7-4	3.8.2-1
ncurses-devel	5.9-13.20130511	6.3-2
glibc-devel	2.17-111	2.34-112
patch	2.7.1-10	2.7.6-12
redhat-lsb-core	4.1	n/a
readline-devel	7.0-13	8.1-3
libnsl	2.28-36	2.34-70

实验数据及分析

Part1 物理性能测试

- (1)fio测试 在物理机上创建了一块100G大小的虚拟磁盘文件进行测试，参考实验文档中的配置文件中的方式分别测试了4K-R,4K-W,4K-RR,4K-RW情况下的读写性能：





通过对比可以看到，BW数据的走向和IOPS几乎一致，都是4K-R的读取性能最佳，4K-RR的读取性能次之。综合来开，两组顺序读写的性能都要分别好于随机读写。

- (2)跨NUMA节点内存访问性能测试

  参考numactl测试numa间读写速率的方式使用numactl测试跨节点内存访问性能。

```
[root@kp004 github]# numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
node 0 size: 63873 MB
node 0 free: 50598 MB
node 1 cpus: 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
node 1 size: 64507 MB
node 1 free: 60017 MB
node 2 cpus: 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
node 2 size: 64507 MB
node 2 free: 56136 MB
node 3 cpus: 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127
node 3 size: 62970 MB
node 3 free: 57536 MB
node distances:
node   0   1   2   3
  0:  10  16  32  33
  1:  16  10  25  32
  2:  32  25  10  16
  3:  33  32  16  10
```

使用numactl --hardware可以看到，node0到各个node的distances分别是10， 16， 32， 33

- 跨内存写入测试

```
[root@kp004 github]# numactl --cpubind=0 --membind=0 dd if=/dev/zero of=/dev/shm/A bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 0.634476 s, 1.7 GB/s
[root@kp004 github]# numactl --cpubind=0 --membind=1 dd if=/dev/zero of=/dev/shm/A bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 0.7171 s, 1.5 GB/s
[root@kp004 github]# numactl --cpubind=0 --membind=2 dd if=/dev/zero of=/dev/shm/A bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 0.967445 s, 1.1 GB/s
[root@kp004 github]# numactl --cpubind=0 --membind=3 dd if=/dev/zero of=/dev/shm/A bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 1.03089 s, 1.0 GB/s
```

分别使用node0上的CPU向不同node上的内存写入1G数据，速度大小分别是1.7, 1.5, 1.1, 1.0（GB/s）,可以看到写入速度与距离大致成一个反比的关系。

- 跨内存读取测试

```
[root@kp004 github]# numactl --cpubind=0 --membind=3 dd if=/dev/shm/A of=/dev/null bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 1.03279 s, 1.0 GB/s
[root@kp004 github]# numactl --cpubind=1 --membind=3 dd if=/dev/shm/A of=/dev/null bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 1.0653 s, 1.0 GB/s
[root@kp004 github]# numactl --cpubind=2 --membind=3 dd if=/dev/shm/A of=/dev/null bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 0.509148 s, 2.1 GB/s
[root@kp004 github]# numactl --cpubind=3 --membind=3 dd if=/dev/shm/A of=/dev/null bs=1M count=1024
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB, 1.0 GiB) copied, 0.446014 s, 2.4 GB/s
```

分别使用不同node上的CPU，向node3上的内存读取1G数据（/dev/shm/A最后写在node3上），读取速度分别是 1.0, 1/0, 2.1, 2.4（GB/s）测试结果反过来的原因是因为从最远的节点开始进行内存读取测试，此时应该参照的是node3到各个节点的distances。结果基本与写入测试的结果相吻合，并且不同node之间的读取速度差异比写入速度更大，符合参考方法里的测试结果。

- 跨内存读写交叉测试

首先分别将A,B,C,D写入node0， node1， node2和node3,接着分别使用不同node上的CPU和不同node上的内存的组合分别访问A,B,C,D，得到结果如下。其中两处1000+的数据为测试结果为1.0GB/s的数据，详情见log文件：

A,B,C,D（MB/s）	**cpu0**	**cpu1**	**cpu2**	**cpu3**
mem0	975，951，675，664	891，917，624，614	589，580，800，774	553，542，735，779
mem1	916，884，647，640	982，1000+，665，656	580，567，774，755	544，533，714，754
mem2	800，772，578，569	733，767，545，539	694，678，995，985	634，622，901，928
mem3	772，744，564，556	707，742，531，526	663，646，934，907	671，654，986，1000+

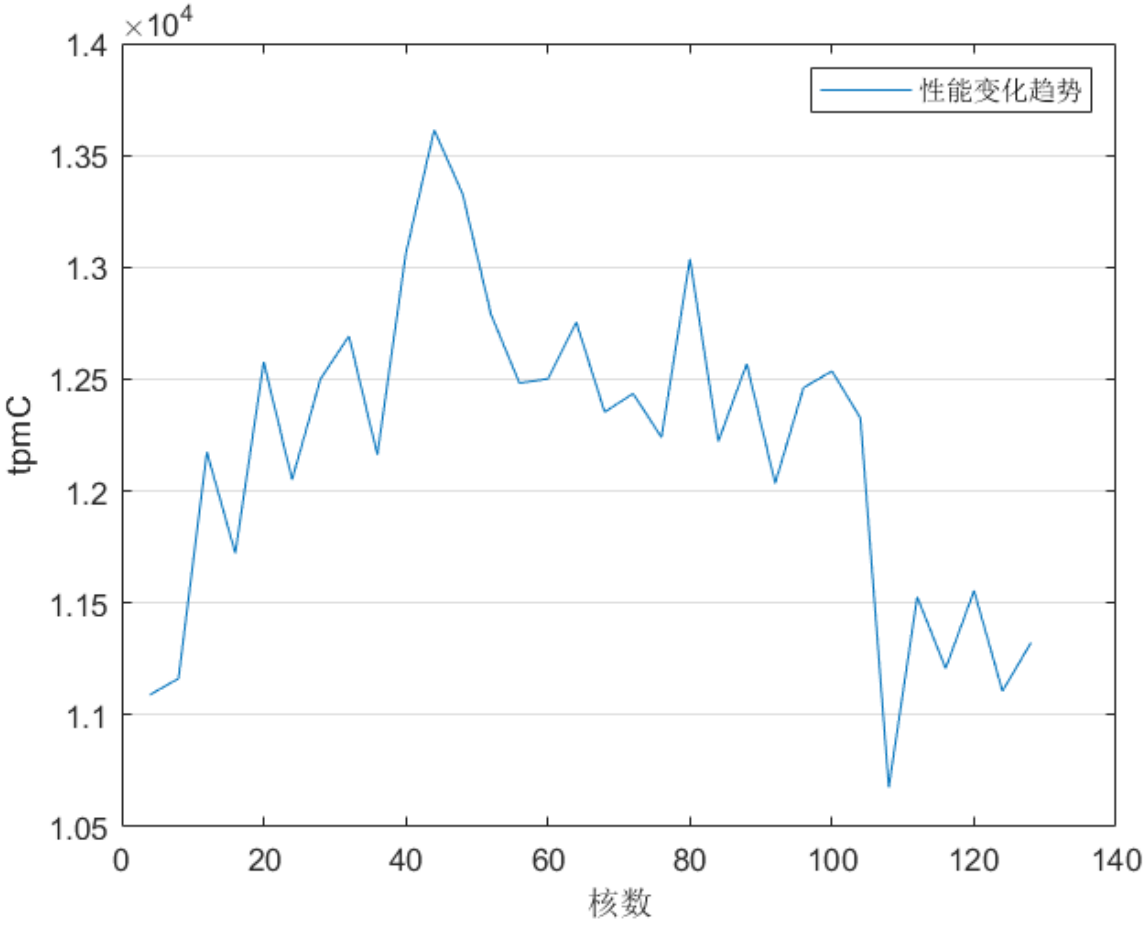
测试结果基本符合单独进行读写测试的结果，在CPU不变的时候，从自己的内存上读数据要比从其他内存上读数据要快得多，而读内存的node固定时，写在自己内存上的速度快于写在其他内存上，速度的大小基本也和node之间的距离成反比关系。

Part2 基础环境下的测试

- (1)跑TPC-C得出tpmC值

```
Term-00, Running Average tpmTOTAL: 25138.38    Current tpmTOTAL: 834864    Memory Usage: 275MB / 2080MB
10:52:07,985 [Thread-121] INFO   jTPCC : Term-00,
10:52:07,987 [Thread-121] INFO   jTPCC : Term-00,
10:52:07,989 [Thread-121] INFO   jTPCC : Term-00, Measured tpmC (NewOrders) = 11321.64
10:52:07,990 [Thread-121] INFO   jTPCC : Term-00, Measured tpmTOTAL = 25117.85
10:52:07,990 [Thread-121] INFO   jTPCC : Term-00, Session Start     = 2023-11-24 10:47:05
10:52:07,990 [Thread-121] INFO   jTPCC : Term-00, Session End       = 2023-11-24 10:52:07
10:52:07,990 [Thread-121] INFO   jTPCC : Term-00, Transaction Count = 126737
```

- (2)单核多核测试结果分析



从图中不难发现，在核数为4-44（步长为4）阶段时，numa架构随核心数增加的可扩展性较好；当核数大于44时，可扩展性下降。

另外，I/O应该不是当前实验的性能瓶颈。我们通过generateReport.sh生成磁盘sda的IOPS图，发现在所有情况下最大的IOPS仅为1w-2w，而Part1中随机写测试出的IOPS为8w左右，两者相差较大，因此，磁盘I/O应该不是瓶颈。

Part3 numa-aware调优

1. 分析PGPROC数据结构及其相关操作函数在TPC-C负载下的执行特点。

- 请你学习主进程中PGPROC类型对象的分配方式，并分析其在内存分布上有什么特点，在NUMA场景下可能会导致怎样的性能问题 主进程中会如下首先预先分配一定数量的PGPROC对象，当有新的进程需要创建时，会从预分配的对象池中获取一个空闲的PGPROC对象，并将其分配给新的进程

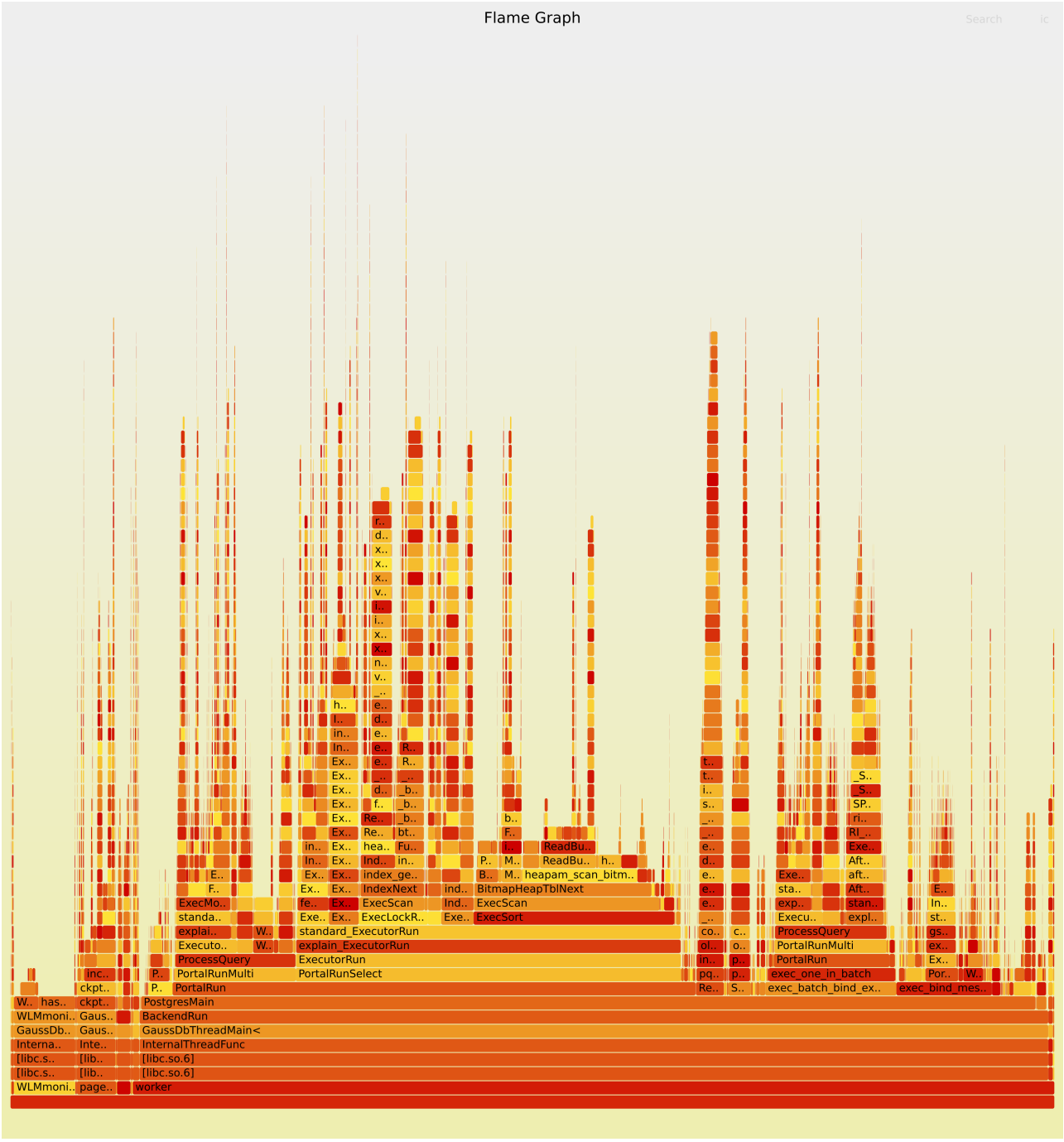
```
    PGPROC *initProcs[MAX_NUMA_NODE] = {0};

    int nNumaNodes = g_instance.shmem_cxt.numaNodeNum;
    /* since myProcLocks is a various array, need palloc actrual size */
    Size actrualPgProcSize = MAXALIGN(offsetof(PGPROC, myProcLocks) +
NUM_LOCK_PARTITIONS * sizeof(SHM_QUEUE)) +
       MAXALIGN(FP_LOCKBIT_NUM * sizeof(uint64)) +
MAXALIGN(FP_LOCK_SLOTS_PER_BACKEND * sizeof(FastPathTag));
    Size fpLockBitsOffset = MAXALIGN(offsetof(PGPROC, myProcLocks) +
```

```
NUM_LOCK_PARTITIONS * sizeof(SHM_QUEUE));
    Size fpRelIdOffset = fpLockBitsOffset + MAXALIGN(FP_LOCKBIT_NUM *
sizeof(uint64));
#ifdef __USE_NUMA
    if (nNumaNodes > 1) {
        ereport(INFO, (errmsg("InitProcGlobal nNumaNodes: %d, inheritThreadPool:
%d, groupNum: %d",
                              nNumaNodes, g_instance.numa_cxt.inheritThreadPool,
                              (g_threadPoolControler ? g_threadPoolControler-
>GetGroupNum() : 0))));

        int groupProcCount = (TotalProcs + nNumaNodes - 1) / nNumaNodes;
        size_t allocSize = groupProcCount * actrualPgProcSize;
        for (int nodeNo = 0; nodeNo < nNumaNodes; nodeNo++) {
            initProcs[nodeNo] = (PGPROC *)numa_alloc_onnode(allocSize, nodeNo);
            if (!initProcs[nodeNo]) {
                ereport(FATAL, (errcode(ERRCODE_OUT_OF_MEMORY),
                                errmsg("InitProcGlobal NUMA memory allocation in
node %d failed.", nodeNo)));
            }
            add_numa_alloc_info(initProcs[nodeNo], allocSize);
            int ret = memset_s(initProcs[nodeNo], allocSize, 0, allocSize);
            securec_check_c(ret, "\0", "\0");
        }
    } else {
#endif
        if (needPalloc) {
            initProcs[0] = (PGPROC *)CACHELINEALIGN(palloc0(TotalProcs *
actrualPgProcSize + PG_CACHE_LINE_SIZE));
        } else {
            initProcs[0] = g_instance.proc_base->allProcs[0];
            errno_t rc = memset_s(initProcs[0], TotalProcs * actrualPgProcSize, 0,
TotalProcs * actrualPgProcSize);
            securec_check(rc, "", "");
        }
        if (!initProcs[0]) {
            ereport(FATAL, (errcode(ERRCODE_OUT_OF_MEMORY), errmsg("out of shared
memory")));
        }
#ifdef __USE_NUMA
    }
#endif
    if (needPalloc) {
        procs = (PGPROC **)CACHELINEALIGN(palloc0(TotalProcs * sizeof(PGPROC *) +
PG_CACHE_LINE_SIZE));
        g_instance.proc_base->allProcs = procs;
    } else {
        errno_t rc = memset_s(g_instance.proc_base->allProcs, TotalProcs *
sizeof(PGPROC *), 0,
            TotalProcs * sizeof(PGPROC *));
        securec_check(rc, "", "");
        procs = g_instance.proc_base->allProcs;
    }
```

```
    g_instance.proc_base->allProcCount = TotalProcs;
    g_instance.proc_base->allNonPreparedProcCount =
g_instance.shmem_cxt.MaxBackends +
                                                    NUM_CMAGENT_PROCS +
NUM_AUXILIARY_PROCS +
                                                    NUM_DCF_CALLBACK_PROCS +
NUM_DMS_CALLBACK_PROCS;
    if (procs == NULL)
        ereport(FATAL, (errcode(ERRCODE_OUT_OF_MEMORY), errmsg("out of shared
memory")));

    for (i = 0; (unsigned int)(i) < TotalProcs; i++) { /* set proc pointer to
actural position */
        procs[i] = (PGPROC *)((char*)(initProcs[i % nNumaNodes]) + (i /
nNumaNodes) * actrualPgProcSize);
    }
```

在非numa优化的情况下，内存分布是连续的，可以提高内存访问的效率，但在numa场景下会导致远程内存访问的开销增加。

- 请你学习PGPROC类型对象相关操作函数的功能，并在TPC-C负载下，通过火焰图分析PGPROC相关的操作函数执行的CPU时间占比，指出相关操作中性能瓶颈所在



根据火焰图，执行器操作占比最大，在Worker中，我们主要关注GaussDbThreadMain()，在GaussDbThreadMain()函数中，会初始化线程基本信息，接着就是调用不同线程的main函数。在这一过程中会使用CreateSharedMemoryAndSemaphores()完成共享内存初始化以及共享数据结构的创建，其中的InitProcGlobal()负责初始化进程表，申请PGPROC数组。InitProcess()中的 GetProcFromFreeList()负责为新的线程分配PGPROC结构的内存。我们将通过对这一过程中PGPROC的分配进行优化来减少GaussDbThreadMain对性能的限制。

2. 设计并实现NUMA-aware的PGPROC结构体分配方式。

目前openGuass已经通过numa_alloc_onnode在预分配时在各个NUMA节点上分配一定数量的PGPROC结构体，线程中分配则是通过遍历一个空闲PGPROC指针链表，找到其中在当前线程的NUMA节点下创建的PGPROC结构体来分配给这个线程。我们的思路是使用和NUMA节点数相同的空闲链表，每个链表管理各个NUMA节点的空闲PGPROC结构体，分配时只需要用对应的空闲链表，如果用完了则使用其它空闲链表，代码修改如下：

proc.h

```
typedef struct PROC_HDR {
...
    PGPROC** freeProcs;
...
} PROC_HDR;
```

proc.cpp

```
void InitProcGlobal(void)
{
...
    int nNumaNodes = g_instance.shmem_cxt.numaNodeNum;
    g_instance.proc_base->freeProcs = (PGPROC **)CACHELINEALIGN(palloc0(nNumaNodes
* sizeof(PGPROC *)));
...
    if (i < g_instance.shmem_cxt.MaxConnections + thread_pool_stream_proc_num +
AUXILIARY_BACKENDS) {
            /* PGPROC for normal backend and auxiliary backend, add to freeProcs
list */
            procs[i]->links.next = (SHM_QUEUE *)g_instance.proc_base->freeProcs[i
% nNumaNodes];
            g_instance.proc_base->freeProcs[i % nNumaNodes] = procs[i];
...
}
PGPROC *GetFreeProc()
{
    if (!g_instance.proc_base->freeProcs) {
        return NULL;
    }
    int nNumaNodes = g_instance.shmem_cxt.numaNodeNum;
    int numaNodeNo = 0;
    if (t_thrd.threadpool_cxt.worker && g_instance.numa_cxt.inheritThreadPool)
        numaNodeNo = t_thrd.threadpool_cxt.worker->GetGroup()->GetNumaId();
    int realNodeNo = numaNodeNo;
    PGPROC *current = NULL;
    for (int i = 0; i < nNumaNodes; i ++) {
        current = g_instance.proc_base->freeProcs[realNodeNo];
        if (current) {
            g_instance.proc_base->freeProcs[realNodeNo] = (PGPROC *)current-
>links.next;
            return current;
        }
        realNodeNo = (realNodeNo + 1) % nNumaNodes;
    }

    return NULL;
}
bool HaveNFreeProcs(int n)
{
    PGPROC* proc = NULL;
```

```
    ProcBaseLockAccquire(&g_instance.proc_base_mutex_lock);

    if (u_sess->libpq_cxt.IsConnFromCmAgent) {
        proc = g_instance.proc_base->cmAgentFreeProcs;
        while (n > 0 && proc != NULL) {
            proc = (PGPROC*)proc->links.next;
            n--;
        }
    } else {
        int nNumaNodes = g_instance.shmem_cxt.numaNodeNum;
        for (int i = 0; i < nNumaNodes && n > 0; i ++) {
            proc = g_instance.proc_base->freeProcs[i];
            while (n > 0 && proc != NULL) {
                proc = (PGPROC*)proc->links.next;
                n--;
            }
        }
    }

    ProcBaseLockRelease(&g_instance.proc_base_mutex_lock);

    return (n <= 0);
}
static void ProcPutBackToFreeList()
{
...
    } else {
        int numaNodeNo = t_thrd.proc->nodeno;
        t_thrd.proc->links.next = (SHM_QUEUE*)g_instance.proc_base-
>freeProcs[numaNodeNo];
        g_instance.proc_base->freeProcs[numaNodeNo] = t_thrd.proc;
        if (t_thrd.role == WORKER && u_sess->proc_cxt.PassConnLimit) {
            SpinLockAcquire(&g_instance.conn_cxt.ConnCountLock);
            g_instance.conn_cxt.CurConnCount--;
            Assert(g_instance.conn_cxt.CurConnCount >= 0);
            SpinLockRelease(&g_instance.conn_cxt.ConnCountLock);
        }
    }
}
```

3. 性能比较

	不使用NUMA优化	**自带NUMA优化**	**我们优化**
tpmC	9304.24	10227.33	10862.51

附录

参数

本次实验使用了虚拟磁盘文件进行测试，创建方法同实验指导文档

fio测试所使用的命令如下：

```
$ dd if=/dev/zero of=bdev.img bs=1G count=100
```

fio测试使用的bench.conf同实验指导文档，使用上面创建的bdev.img进行实验，具体内容在实验截图中可以看到，此处不再进行展示。

跨NUMA节点内存访问性能测试中使用的测试脚本如下：

```
#!/bin/bash
for cpus in {0..3}; do
  for mems in {0..3}; do
    for i in A B C D; do
      numactl --cpubind=$cpus --membind=$mems dd if=/dev/shm/$i
of=/dev/shm/node$mems bs=1M count=1024
    #   echo "Core binding: $cpus, Memory binding: $mems, Input file: $i"
    done
  done
done
```

最终测试结果保存在result.log中，详情见附件

openGauss编译脚本如下

```
#!/bin/bash

cd openGauss-server
export CODE_BASE=/home/CloudOS/openGauss-server     # openGauss-server的路径
export BINARYLIBS=/home/CloudOS/binarylibs    # binarylibs的路径
export GAUSSHOME=$CODE_BASE/dest/
export GCC_PATH=$BINARYLIBS/buildtools/gcc7.3/
export CC=$GCC_PATH/gcc/bin/gcc
export CXX=$GCC_PATH/gcc/bin/g++
export
LD_LIBRARY_PATH=$GAUSSHOME/lib:$GCC_PATH/gcc/lib64:$GCC_PATH/isl/lib:$GCC_PATH/mpc
/lib/:$GCC_PATH/mpfr/lib/:$GCC_PATH/gmp/lib/:$LD_LIBRARY_PATH
export PATH=$GAUSSHOME/bin:$GCC_PATH/gcc/bin:$PATH

make clean
./configure --gcc-version=7.3.0 CC=g++ CFLAGS="-O2 -g3 -D__USE_NUMA" --
prefix=$GAUSSHOME --3rd=$BINARYLIBS --enable-thread-safety --with-readline --
without-zlib
make -sj
make install -sj
chown -R omm:dbgrp /home/CloudOS/openGauss-server
```

benchmarkSQL配置文件为`2_code/props.opengauss.1000w` openGauss配置文件为`2_code/postgresql.conf`

贡献度情况

任务分工：

- 环境配置：洪墅霖、游克垚
- Part1 物理性能测试：洪墅霖
- Part2 基础环境下的测试：李天策
- Part3 numa-aware调优 ：高鹏轩、游克垚

贡献度：

- 游克垚：30%
- 洪墅霖：25%
- 高鹏轩：22.5%
- 李天策：22.5%