Linpack测试包括三类，Linpack100、Linpack1000和HPL。Linpack100求解规模为100阶的稠密线性代数方程组，它只允许采用编译优化选项进行优化，不得更改代码，甚至代码中的注释也不得修改。Linpack1000要求求解规模为1000阶的线性代数方程组，达到指定的精度要求，可以在不改变计算量的前提下做算法和代码上做优化。HPL即High Performance Linpack，也叫高度[并行计算](https://baike.baidu.com/item/%E5%B9%B6%E8%A1%8C%E8%AE%A1%E7%AE%97" \t "https://baike.baidu.com/item/linpack/_blank)[基准测试](https://baike.baidu.com/item/%E5%9F%BA%E5%87%86%E6%B5%8B%E8%AF%95)，它对[数组](https://baike.baidu.com/item/%E6%95%B0%E7%BB%84)大小N没有限制，求解问题的规模可以改变，除基本算法（计算量）不可改变外，可以采用其它任何优化方法。

HPL是针对现代并行计算机提出的测试方式。用户在不修改任意[测试程序](https://baike.baidu.com/item/%E6%B5%8B%E8%AF%95%E7%A8%8B%E5%BA%8F" \t "https://baike.baidu.com/item/linpack/_blank)的基础上，可以调节问题规模大小N([矩阵](https://baike.baidu.com/item/%E7%9F%A9%E9%98%B5)大小)、使用到的CPU数目、使用各种优化方法等来执行该测试程序，以获取最佳的性能。HPL采用[高斯消元法](https://baike.baidu.com/item/%E9%AB%98%E6%96%AF%E6%B6%88%E5%85%83%E6%B3%95" \t "https://baike.baidu.com/item/linpack/_blank)求解[线性方程组](https://baike.baidu.com/item/%E7%BA%BF%E6%80%A7%E6%96%B9%E7%A8%8B%E7%BB%84/5904308)。当求解问题规模为N时，[浮点运算](https://baike.baidu.com/item/%E6%B5%AE%E7%82%B9%E8%BF%90%E7%AE%97" \t "https://baike.baidu.com/item/linpack/_blank)次数为(2/3 \* N^3－2\*N^2)。因此，只要给出问题规模N，测得系统计算时间T，峰值=计算量(2/3 \* N^3－2\*N^2)/计算时间T，测试结果以浮点运算每秒（[Flops](https://baike.baidu.com/item/Flops/989494" \t "https://baike.baidu.com/item/linpack/_blank)）给出。

衡量计算机性能的一个重要指标就是计算峰值，例如浮点计算峰值，它是指计算机每秒钟能完成的浮点计算最大次数。包括理论浮点峰值和实测浮点峰值：

理论浮点峰值是该计算机理论上能达到的每秒钟能完成浮点计算最大次数，它主要是由CPU的主频决定的，

理论浮点峰值=[CPU主频](https://baike.baidu.com/item/CPU%E4%B8%BB%E9%A2%91/3519849)×CPU每个[时钟周期](https://baike.baidu.com/item/%E6%97%B6%E9%92%9F%E5%91%A8%E6%9C%9F/1545064" \t "https://baike.baidu.com/item/linpack/_blank)执行[浮点运算](https://baike.baidu.com/item/%E6%B5%AE%E7%82%B9%E8%BF%90%E7%AE%97)的次数×系统中[CPU核心](https://baike.baidu.com/item/CPU%E6%A0%B8%E5%BF%83/3030437" \t "https://baike.baidu.com/item/linpack/_blank)数目。

实测浮点峰值是指Linpack测试值，也就是说在这台机器上运行Linpack[测试程序](https://baike.baidu.com/item/%E6%B5%8B%E8%AF%95%E7%A8%8B%E5%BA%8F" \t "https://baike.baidu.com/item/linpack/_blank)，通过各种调优方法得到的最优的测试结果。实际上在实际程序运行过程中，几乎不可能达到实测浮点峰值，更不用说达到理论浮点峰值了。这两个值只是作为衡量机器性能的一个指标，用来表明机器处理能力的一个标尺和潜能的度量。

安装HPL之前需要配置好：

1. [GCC/Fortran77 编译器](https://www.cnblogs.com/zhyantao/p/10614244.html)
2. openblas库
3. mpich并行环境
4. gcc编译器之前已经安装好了，就使用gcc-9.2.0.
5. 安装openblas库：
   1. 下载openblas压缩包并上传
   2. 更改名字解压 tar -zxvf OpenBLAS-0.3.17-version.tar.gz
   3. cd OpenBLAS-0.3.17
   4. make
   5. make install PREFIX=/home/chenrenxing/OpenBLAS
   6. 配置环境变量：export LD\_LIBRARY\_PATH=/home/chenrenxing/OpenBLAS/lib:$LD\_LIBRARY\_PATH
6. 安装mpich-3.4.2
   1. wget <http://www.mpich.org/static/downloads/3.4.2/mpich-3.4.2.tar.gz>
   2. cd mpich-3.4.2
   3. ./configure --with-device=ch4:ofi --prefix=/home/chenrenxing/mipch3.4.2
   4. make -j6
   5. make install
   6. 配置环境变量：

export PATH=/home/chenrenxing/mipch3.4.2/bin/:$PATH

export LD\_LIBRARY\_PATH=/home/chenrenxing/mpich3.4.2/lib:$LD\_LIBRARY\_PATH

export MANPATH=/home/chenrenxing/mpich3.4.2/share/man:$MANPATH

下载HPL：

1. 下载hpl-2.3压缩包并上传
2. tar xf hpl-2.3.tar.gz
3. cd hpl-2.3/setup
4. sh make\_generic
5. cd ..
6. cp setup/Make.UNKNOWN Make.rpi
7. 打开Make.rpi文件修改如下内容：

ARCH = rpi

TOPdir = /home/chenrenxing/hpl-2.3

INCdir = $(TOPdir)/include

BINdir = $(TOPdir)/bin/$(ARCH)

LIBdir = $(TOPdir)/lib/$(ARCH)

MPdir = /home/chenrenxing/mipch3.4.2

MPinc = -I$(MPdir)/include

MPlib = -L$(MPdir)/lib

LAdir = /home/chenrenxing/OpenBLAS

LAinc =

LAlib = $(LAdir)/libopenblas.a $(LAdir)/libopenblas\_haswellp-r0.3.17.a

CC = /home/chenrenxing/mipch3.4.2/bin/mpicc

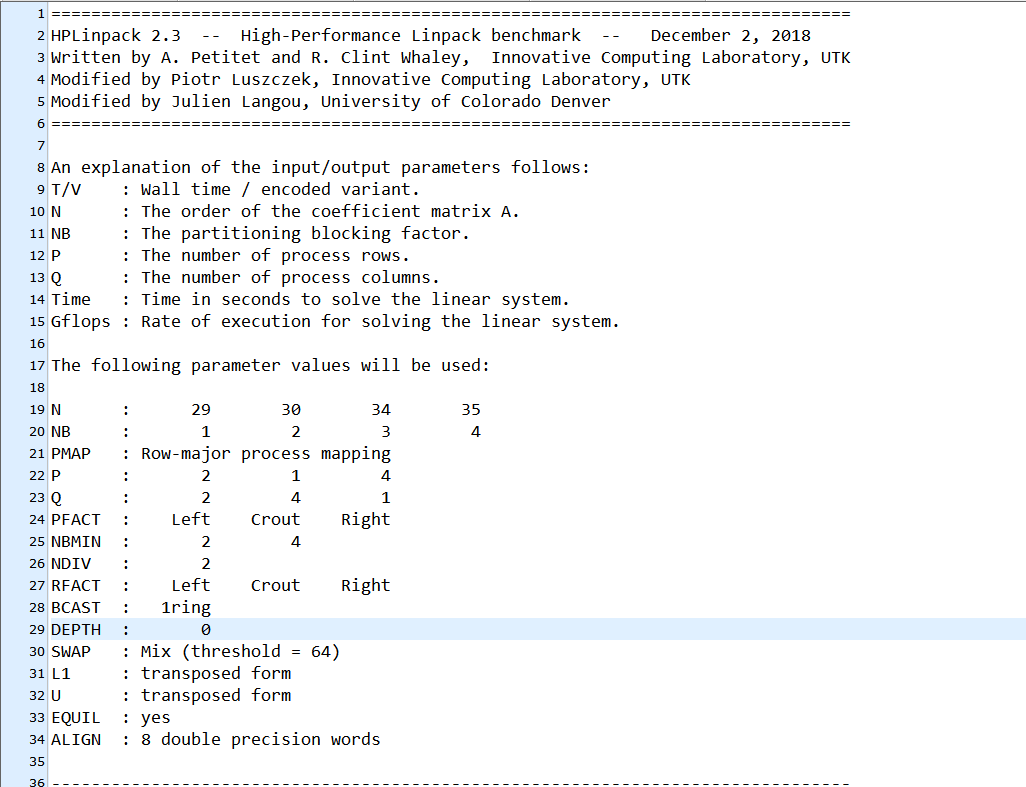
CCNOOPT = $(HPL\_DEFS)

CCFLAGS = $(HPL\_DEFS) -fomit-frame-pointer -O3 -funroll-loops -W -Wall -pthread

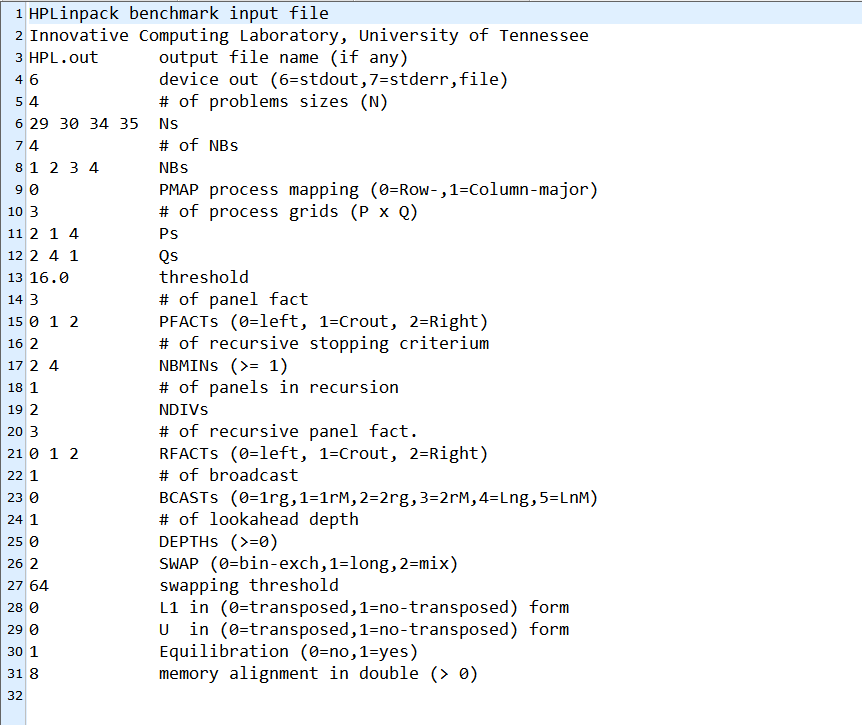
LINKER = /home/chenrenxing/mipch3.4.2/bin/mpif77

LINKFLAGS = $(CCFLAGS)

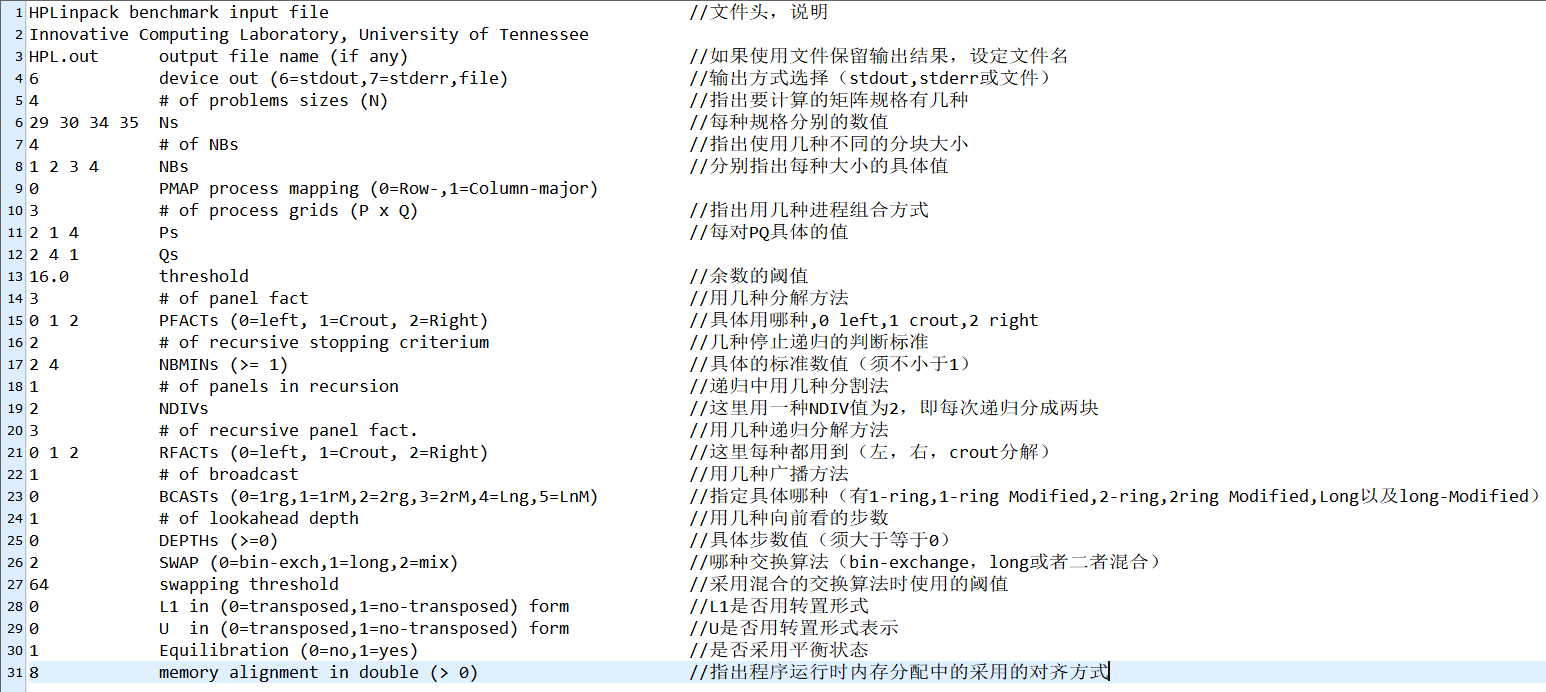
1. make arch=rpi
2. 初步测试:
   1. cd /home/chenrenxing/hpl-2.3/bin/rpi
   2. mpirun -np 4 ./xhpl > HPL-Benchmark.txt 内容如下：



初始的HPL.dat文件：



文件说明：



HPL优化：

主要是对以下几个参数进行修改：

1、Ns: N = ((节点数\*内存／8)\*80)^1/2

Ns表示求解线性方程组Ax＝b中矩阵A的规模（N）。矩阵的规模N越大，有效计算所占的比例也越大，系统浮点处理性能也就越高；但与此同时，矩阵规模N的增加会导致内存消耗量的增加，一旦系统实际内存空间不足，使用缓存，性能会大幅度降低。

考虑到操作系统本身需要占用一定的内存，除了矩阵A（N×N）之外，HPL还有其它的内存开销，另外通信也需要占用一些缓存（具体占用的大小视不同的MPI而定）。具体N最优选择还跟实际的软硬件环境密切相关。当整个系统规模较小、节点数较少、每个节点的内存较大时，N可以选择大一点。当整个系统规模较大、节点数较多、每个节点的内存较小时是，N可以选择大一点。

问题规模 N的选择：问题规模的大小也决定了计算量的大小，N越大，需要花费的时间越长，需要占用的内存空间就越多。如果选择的 N超过一定值，内存就不够用而发生磁盘交换，从而大大降低系统的计算效率。所以选择 N之前应该首先按照前面所提到公式估算一下。在这个临界值之下，N越大，性能指标就越高，是因为N变大了，那么计算量相对于通信量的开销也就增大，即 N 较大时计算量／通信量比也较大。

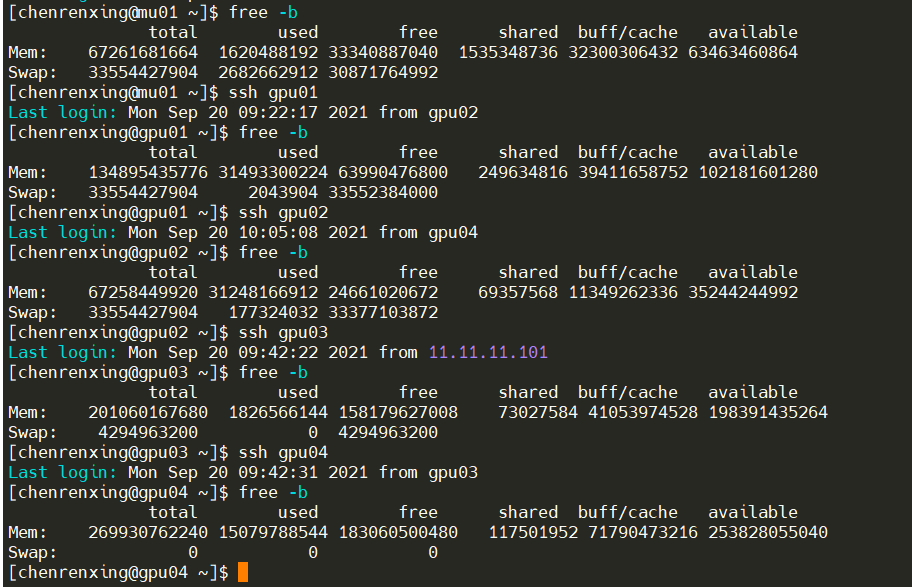
1. NB

数据块大小NB的选择 从数据分布的角度上来看， NB越小，分块就越多，也就越有利于各进程的负载平衡；但从计算的角度上来看，太小的NB，会限制系统的计算性能，因为在最高一级的存储结构中数据重用很少，而且需要交换 的消息增多。NB的最佳值跟N的大小有关，也跟整个系统 的计算／通信性能比有关。

衡量 NB是否最佳的根本原因还是看它产生的通信开销是否达到最小，从通信角度来看，NB太小会大大地增加进程间的通信置，降低计算所占比重，而 NB太大，影响各个处理器的负载平衡，同样也会增加通信开销。

1. 处理器网格分布的选择（P和Q） 当P和Q比较接近且P小于Q的时候，效率最高，原因如下：P和 Q 接近时有利于进程之间的负载平衡，行列方向通信能够保持平衡，否则可能同一对处理器之间要进行大量消息的通信；在此前提下，P要稍小于Q，因为根据 Linpack的算法，列向通信量要大于行向通信量，使列向的进程稍多一些可以减少通信开销。

各个节点的内存：



安装openmpi-4.1.1

1. 下载安装包并上传
2. tar -zxvf openmpi-4.1.1.tar.gz
3. cd openmpi-4.1.1
4. ./configure --prefix=/home/chenrenxing/openmpi4.1.1
5. make && make install
6. 配置环境变量：

MPI\_HOME=/home/chenrenxing/openmpi4.1.1

export PATH=${MPI\_HOME}/bin:$PATH

export LD\_LIBRARY\_PATH=${MPI\_HOME}/lib:$LD\_LIBRARY\_PATH

export MANPATH=${MPI\_HOME}/share/man:$MANPATH

安装mkl

1. 先从官网上面下载以.sh结尾的脚本文件 [下载英特尔® oneAPI 基础工具包 (intel.com)](https://software.intel.com/content/www/us/en/develop/tools/oneapi/base-toolkit/download.html?operatingsystem=linux&distributions=webdownload&options=offline)，并上传到服务器
2. 执行命令：bash l\_BaseKit\_p\_2021.3.0.3219\_offline.sh(下载的脚本文件)
3. 再选择Down Only，如果全部选择就会占用太多的内存，没有必要，在选择不需要的软件按空格键取消，只留下mkl库，之后选择下载。
4. 在个人目录下生成intel文件，其中有刚生成的文件intel.oneapi.lin.BaseKit.package,v=2021.3.0-3219，进入文件中执行命令：./install.sh --download-dir=/home/chenrenxing/mkl。最后选择下载即可。

安装hpcg:

1. git clone <https://github.com/hpcg-benchmark/hpcg.git>
2. cd hpcg/setup
3. 修改Make.Linux\_MPI文件

TOPdir = /home/chenrenxing/hpcg

MPdir =/home/chenrenxing/mipch3.4.2

MPinc =-I$(MPdir)/include

MPlib =-L$(MPdir)/lib

CXX = /home/chenrenxing/mipch3.4.2/bin/mpicxx

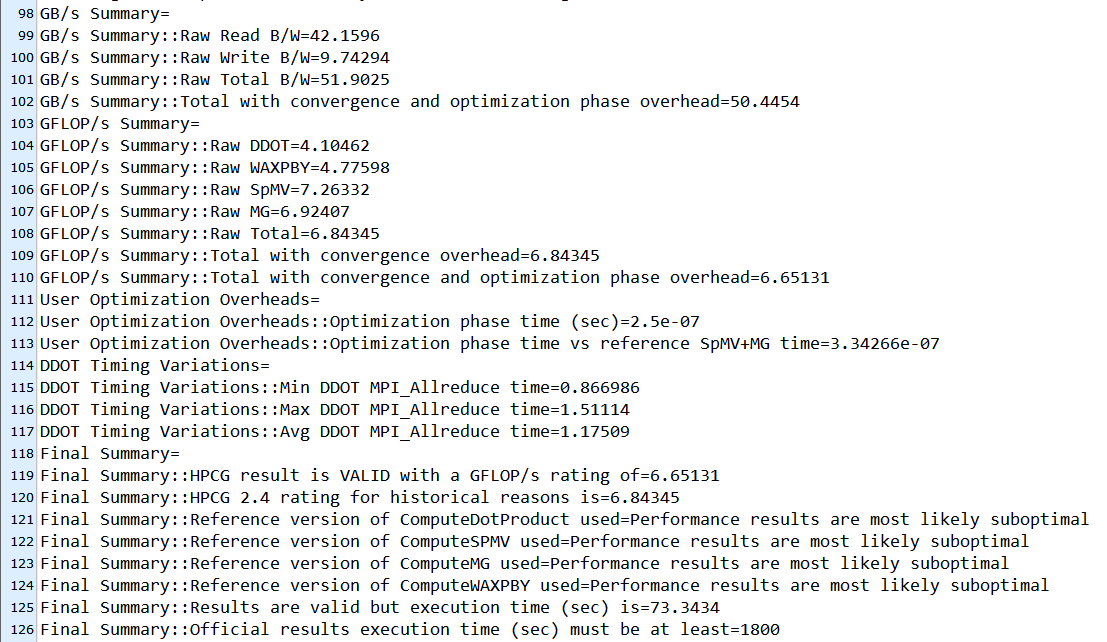
其余不变

1. mkdir build && cd build
2. /home/chenrenxing/hpcg/configure Linux\_MPI
3. make
4. 运行测试：

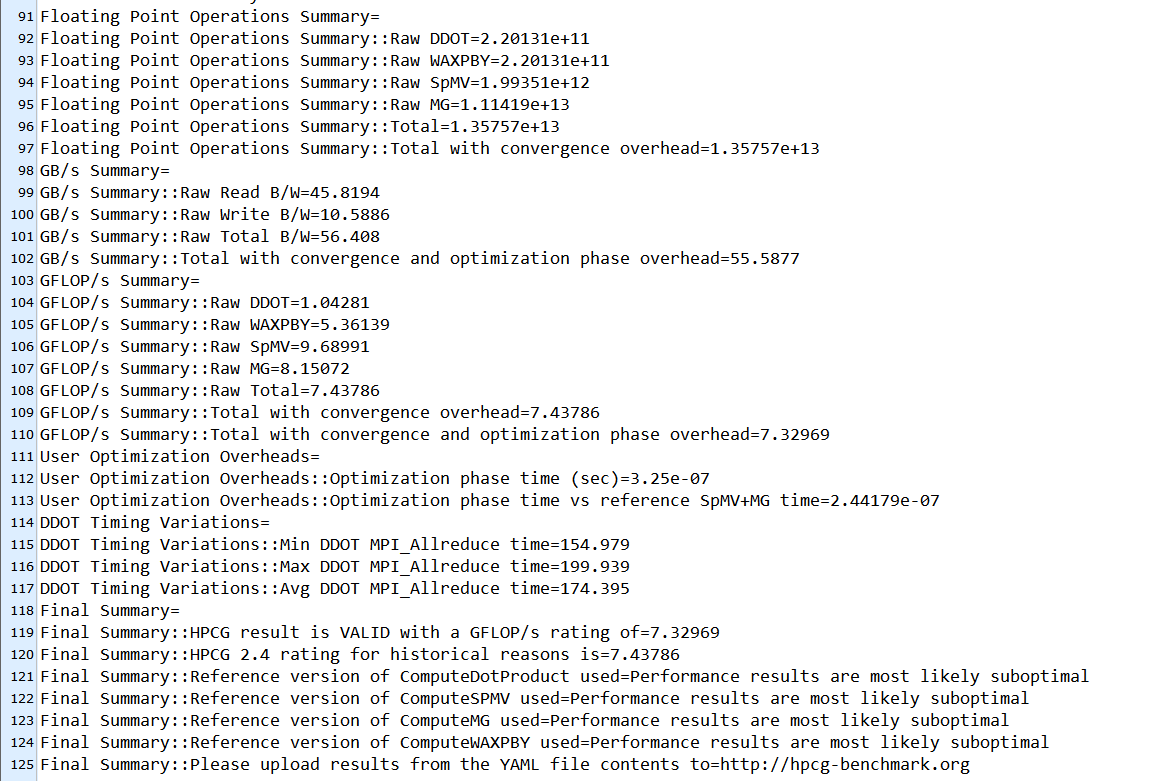
cd bin

mpirun -np 12 ./xhpcg

问题规模104 104 104，运行时间60s：



问题规模104 104 104，运行时间1800：



HPCG优化问题：

1. 执行问题规模的选择和运行时间的选择，hpcg官方规定，hpcg运行时间必须要1800s才能得到一个正式的结果。
2. 对于其他的mpi还没有进行尝试。
3. # 总核数 = 物理CPU个数 X 每颗物理CPU的核数
4. # 总逻辑CPU数 = 物理CPU个数 X 每颗物理CPU的核数 X 超线程数
6. # 查看物理CPU个数
7. cat /proc/cpuinfo| grep "physical id"| sort| uniq| wc -l
9. # 查看每个物理CPU中core的个数(即核数)
10. cat /proc/cpuinfo| grep "cpu cores"| uniq
12. # 查看逻辑CPU的个数
13. cat /proc/cpuinfo| grep "processor"| wc -l
15. #查看CPU信息（型号）
16. cat /proc/cpuinfo | grep name | cut -f2 -d: | uniq -c

nvidia-smi:查看gpu使用情况

hpcg评分是加权 gflop/( 每秒的floating浮点运算) 值，由pcg迭代阶段所执行的操作组成，该值由taken迭代阶段执行。 问题结构的开销时间和改进性能的任何修改都被除以 500迭代( amortization重量) 并添加到运行时。

hpl的优化:

目前统计出四个计算节点的N的规模：gpu01:116144，gpu02:82011，gpu03:141759,

gpu04:164295，但是测试的时候是采用N为63000来测试。目前测试的初步结果如下：

1. 在N、NB、P、Q值都相同的情况下，在gpu04节点上面运行的时间最少，同时测试最大Gflops峰值达到4.9259e+02Gflops。同时在gpu01和gpu02这两个节点上运行的时间和最大Gfops峰值并没有多大的差异，其中gpu03节点运行时间最长，最大Gfops峰值最低。虽然P和Q的值都已经限定，但是在运行的时候，运行时间和最大Gfops峰值好像和本身拥有的最大进程数有关。
2. 在NB设置不同时，分别为128和256，其他参数设置相同，对于运行时间和Gfops峰值NB设置为256效果明显好于NB设置为128。
3. 存在问题1：单个运行测试的效果好于多个运行，比如63000-128中运行时间大概快40秒，运算的浮点数相差了几万亿次。

接下来的尝试：

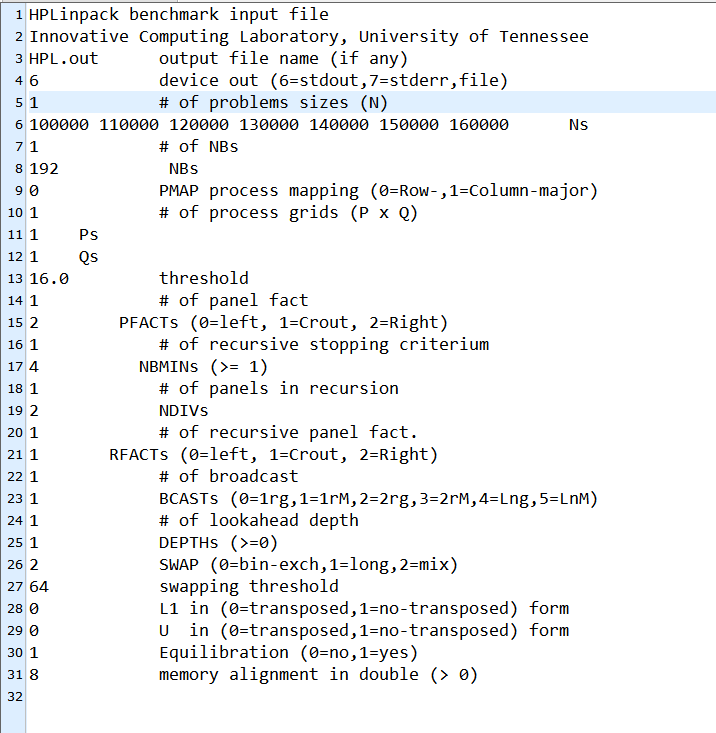
1. 满线程计算浮点数
2. 更改P\*Q数（2\*2、2\*4、2\*5）在此之前需要求出最优N值
3. 更改软件组合，openBLAS+openmpi、mkl+pich、mkl+mipch
4. 软件组合下直接计算最优N值和运算最快的P\*Q

CPU的理论峰值为：2.1(主频)\*40(CPU核数)\*32(2个FMA) = 2688

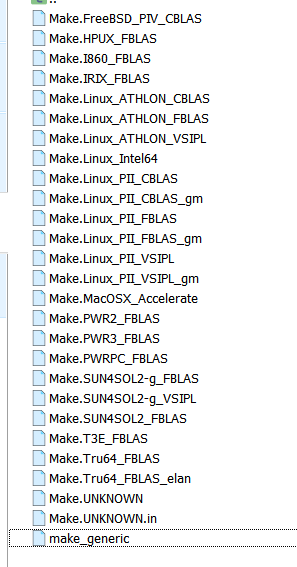
gpu04节点服务器信息：

<https://www.intel.com/content/www/cn/zh/products/sku/192437/intel-xeon-gold-6230-processor-27-5m-cache-2-10-ghz/specifications.html>

修改HPL.dat文件中的参数：



修改文件中的递归分解方法和分解方法的次数，使其计算的次数为1



1. 优化达不到P\*Q的可能：

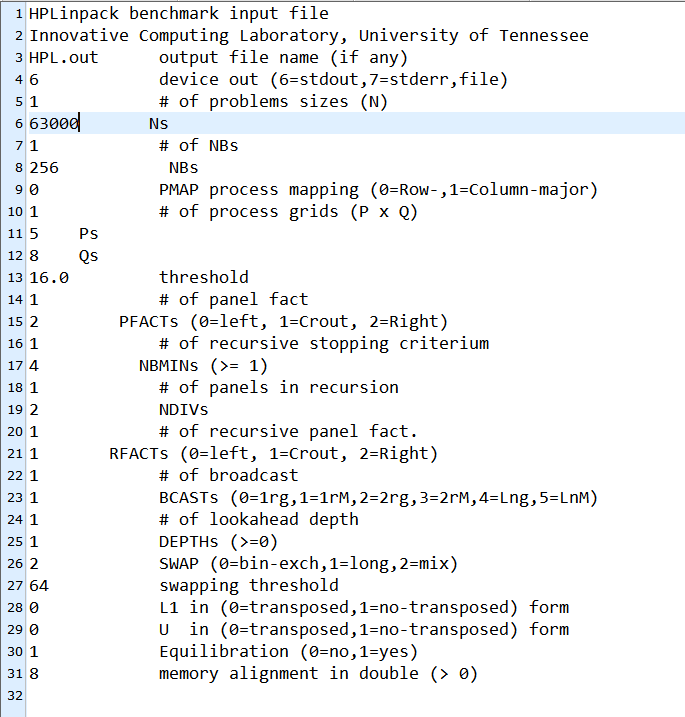
\*在刚开始配置hpl的运行脚本的时候出现问题，可以更改脚本尝试。

\*GPU和CPU两者之间的问题。

\*可能本身就是mipch软件问题

\*mpirun和mpiexec

\*将P\*Q设置为2\*20和4\*10



HPL.dat文件参数理解：

1. 第3行：基准测试输出文件名，默认情况下输出为HPL.out，可以根据个人情况更改，如

mpirun -np 40 ./xhpl > 输出文件名。

1. 第4行：这一行指定输出的位置。该行已格式化，必须以正整数开头，其余无意义。正整数有 3 种选择，6 表示输出将进入标准输出，7 表示输出将进入标准错误。一般情况下选择标准输出。
2. 第5行：这一行指定要执行的问题大小的数量。这个数字应该小于或等于 20，如果这一行写着是3，则只是执行3个N值，如果N值超过3个，3以后的N将不被执行。
3. 第6行，这一行指定了想要运行的问题大小，如果有多个，则按照上面一行规定的个数来执行。N值代表了要求解的矩阵的数量和规模。矩阵大小N越大，一般来说有效计算的比例越大，系统浮点处理性能越高；但如果N值过大就会导致内存消耗的增加，一旦实际系统内存空间不足，使用缓存，性能会大大降低。建议使用公式：N×N×8=（系统总内存)×80%(总内存以字节转换）来计算N值。(如果系统中运行的其他进程很少，可以再适当加大 N)。
4. 第7行，这一行指定要运行的块大小的数量。这个数字应该小于或等于 20，与第四行类似。
5. 第8行，这一行指定了想要运行的块大小。LU分解是线性方程组求解的主要部分，占去的运行时间是O(n)它的计算工作量又主要在更新矩阵A，所以HPL并行的主要任务自然是尽可能地在多处理机上同时对矩阵A的不同部分做更新，这就要求矩阵的分布化。一方面数据按块分布可以考虑每个处理器的内存层次，在单个处理器计算部分更好地利用BLAS3矩阵运算统一做延迟更新，提高运算效率，减少通信；另一方面循环分布可以尽可能合理地在各处理器间分配任务，均衡负载。这就需要对矩阵进行合理的分块。NB的选择与测试平台、数学库等相关，不过有些文档建议NB应尽可能接近Cache行大小，因为这样既能最大限度发挥Cache性能，又减少Cache冲突，还能尽可能地利用BLAS3进行矩阵运算，一般在32~256之间。从数据分布的角度看，NB越小，负载越均衡；从计算角度看，NB太小在很大程度上限制计算性能，因为这样在内存高层几乎没有数据重用，消息的数量也会增加。
6. 第9行，这一行指定 MPI 进程应该如何映射到您平台的节点上。目前有两种可能的映射，即行优先和列优先。当这些节点本身是多处理器计算机时，此功能主要有用。建议使用行优先映射。按HPL文档中介绍，按列的排列方式适用于节点数较多、每个节点内CPU数较少的系统；而按行的排列方式适用于节点数较少、每个节点内CPU数较多的大规模系统。在机群系统上，按列的排列方式的性能远好于按行的排列方式。此处一般选择1。
7. 第10行指定要运行的进程网格数。这个数字应该小于或等于 20。与第4行类似。
8. 第11~12行，这两行指定要在其上运行的每个网格的流程行数和列数。二维处理器网格（P\*Q）的有以下几个要求：
9. P\*Q = 系统CPU数 = 进程数。一般来说一个进程对于一个CPU可以得到最佳性能。对于Intel Xeon来说，关闭超线程可以提高HPL性能。

（2）P和 Q 接近时有利于进程之间的负载平衡，行列方向通信能够保持平衡，否则可能同一对处理器之间要进行大量消息的通信；在此前提下，P要稍小于Q，因为根据 Linpack的算法，列向通信量要大于行向通信量，使列向的进程稍多一些可以减少通信开销。

（3）在集群测试中，P\*Q =系统CPU总核数。

10、 第13行说明测试的精度。这个值就是在做完线性方程组的求解以后，检测求解结果是否正确。若误差在这个值以内就是正确，否则错误。一般而言，若是求解错误，其误差非常大；若正确，则很小。所以没有必要修改此值。

11、第14~21行允许指定算法特征。xhpl 将针对每个问题（N）大小、块(NB)大小、进程网格（P\*Q）组合运行所有可能的组合。panel分解是基于矩阵-矩阵运算的递归式，panel分解的这部分在下面用“RFACT”表示。当当前panel由小于或等于 NBMIN 列组成时，递归停止。在那时候，xhpl 使用基于矩阵向量运算的分解，用“PFACTs”表示。然后经典递归将使用 NDIV=2, NBMIN=1。基本上有 3 种数值等效的 LU 分解算法变体：L、R、C，分别代表Left、Right和Crout。在 HPL 中，可以为 RFACT 和 PFACT 选择其中的每一项。在消元过程中，zHPL采用每次完成NB列的消元，然后更新后面的矩阵。这NB的消元就是L的分解。每次L的分解只在一列处理器中完成。对每一个小矩阵作消元时，都有3种算法：L、R、C，分别代表Left、Right和Crout。在LU分解中，具体的算法很多，测试经验，NDIVs选择2比较理想，NBMINs 4或8都不错。而对于RFACTs和PFACTs，对性能的影响不大。官方文档的建议：

1 # of panel fact

1 PFACTs (0=left, 1=Crout, 2=Right)

2 # of recursive stopping criterium

4 8 NBMINs (>= 1)

1 # of panels in recursion

2 NDIVs

1 # of recursive panel fact.

2 RFACTs (0=left, 1=Crout, 2=Right)

1. 第22~23行说明L（LU分解中A=LU，其中L为下三角矩阵）的横向广播方式，HPL中提供了6种广播方式。其中，前4种适合于快速网络；后两种采用将数据切割后传送的方式，主要适合于速度较慢的网络。目前，机群系统一般采用千兆以太网甚至光纤等高速网络，所以一般不采用后两种方式。一般来说，在小规模系统中，选择0或1；对于大规模系统，选择3。对于广播的直观理解是某个特定进程将数据一一广播到所有的进程，在MPI可以实现一对多的集合通信。
2. 第24~25行说明横向通信的通信深度。这依赖于机器的配置和问题规模的大小。DEPTHs＝0表明将L一次性广播出去，也就是将整个L分解完成以后在一次性广播；DEPTHs＝1表示将L分两次广播；依此类推。小规模系统中，DEPTHs一般选择1或2；对于大规模系统，选择2～5之间。
3. 第26~27行允许指定 HPL 用于所有测试的交换算法。目前有两种可用的交换算法，一种基于“二进制交换”，另一种基于“spread-roll”过程（long）。对于大型问题，最后一个可能更有效。还可以选择混合两种变体，即对小于阈值的列数进行“二进制交换”，然后是“扩展滚动”算法。SWAP="0"，采用二元交换法；SWAP="1"，采用Long法；SWAP="2"，采用混合法。然后在第 27行指定此阈值。

15、第28~29行分别说明L和U的数据存放格式。若选择"transposed"，则采用按列存放，否则按行存放。推荐配置为：

0       L1 in (0=transposed,1=no-transposed) form

0       U  in (0=transposed,1=no-transposed) form

1. 第30行，启用/禁用平衡阶段，很难判断是否应该始终进行平衡。对生成的随机矩阵知之甚少，而且由于开销与可能的增益相比非常小，所以建议将其打开。

16、第31行的值主要为内存地址对齐而设置，用于在内存分配中对齐地址。出于安全考虑，可以选择８运行。

安装impi:

1. wget [http://registrationcenter-download.intel.com/akdlm/irc\_nas/tec/14879/l\_mpi\_2019.1.144.tgz](https://bbs.huaweicloud.com/forum/thread-109547-1-1.html)
2. tar -xzvf l\_mpi\_2019.1.144.tgz
3. cd l\_mpi\_2019.1.144
4. ./install.sh --download\_url =/home/chenrenxing/impi(后面的安装目录可有可不有)
5. 按照出现的界面进行安装，安装后的目录位于intel目录之下
6. 配置环境变量：

export PATH=/home/chenrenxing/intel/compilers\_and\_libraries\_2019.1.144/linux/mpi/intel64/bin/:$PATH

export LD\_LIBRARY\_PATH=/home/chenrenxing/intel/compilers\_and\_libraries\_2019.1.144/linux/mpi/intel64/lib/:$LD\_LIBRARY\_PATH

换库测试HPL(mkl+impi)

1. cd /home/chenrenxing/hpl-2.3/setup
2. cp setup/Make.Linux\_Intel64 Make.intel
3. 打开Make.intel文件修改内容为：

ARCH = intel

TOPdir = /home/chenrenxing/hpl-2.3

INCdir = $(TOPdir)/include

BINdir = $(TOPdir)/bin/$(ARCH)

LIBdir = $(TOPdir)/lib/$(ARCH)

HPLlib = $(LIBdir)/libhpl.a

# MPdir =

/home/chenrenxing/intel/compilers\_and\_libraries\_2019.1.144/linux/mpi/intel64

# MPinc = -I$(MPdir)/include

# MPlib = $(MPdir)/lib/libmpi\_ilp64.a -lfabric

LAdir = /home/chenrenxing/intel/oneapi/mkl/2021.3.0

LAinc = $(LAdir)/include

LAlib = -L$(LAdir)/lib/intel64 $(LAdir)/lib/intel64/libmkl\_intel\_lp64.a \

-Wl,--start-group \

$(LAdir)/lib/intel64/libmkl\_sequential.a \

$(LAdir)/lib/intel64/libmkl\_intel\_thread.a \

$(LAdir)/lib/intel64/libmkl\_core.a \

-Wl,--end-group -L$(LAdir)/lib/intel64 -lpthread -ldl -lm

HPL\_OPTS = -DHPL\_DETAILED\_TIMING -DHPL\_PROGRESS\_REPORT

CC =

/home/chenrenxing/intel/compilers\_and\_libraries\_2019.1.144/linux/mpi/intel64/bin/mpicc

CCNOOPT = $(HPL\_DEFS)

OMP\_DEFS = -openmp

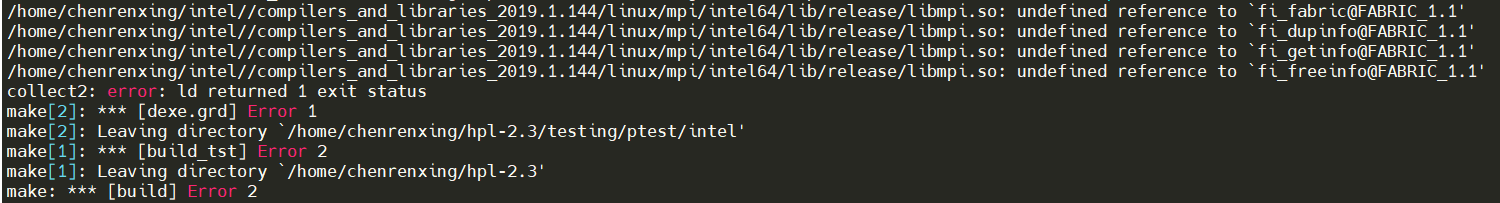
CCFLAGS = $(HPL\_DEFS) -O3 -w -fomit-frame-pointer -z noexecstack -z relro -z now -Wall -funroll-loops -Wrestrict

LINKER = $(CC)

LINKFLAGS = -L$(LAdir)/lib/intel64

1. make arch=intel

出现报错：



原因为再个人目录下libmpi动态库没有引进，于是再环境变量中添加：

export LD\_LIBRARY\_PATH=/home/chenrenxing/intel/compilers\_and\_libraries\_2019.1.144/linux/mpi/intel64/libfabric/lib/:$LD\_LIBRARY\_PATH

激活环境变量之后再次执行make arch=intel即可解决问题。

超线程技术：

超线程技术就是利用特殊的硬件指令，把两个逻辑内核模拟成两个物理芯片，让单个处理器都能使用线程级并行计算．具体讲，就是通过CPU的寄存器构成了两个逻辑处理器，来共享处理器的物理执行单元，并同步进行加、乘、负载等操作．操作系统或者应用软件的多线程可以同时运行于一个HTT处理器上，两个逻辑处理器共享一组处理器执行单元，并行完成加、乘、负载等操作，这样就可以使得运行性能提高，这是因为在同一时间里，应用程序可以使用芯片的不同部分．虽然单线程芯片每秒钟能够处理成千上万条指令，但是在任一时刻只能够对一条指令进行操作．而超线程技术可以使芯片同时进行多线程处理，使芯片性能得到提升。