2020 年

《计算机体系结构》

实验报告

Linpack 性能测试与优化

学院 ： 计算机与通信工程学院

班级 ： 计 172

组号	学号	姓名	贡献比(%)
12	41724051	金玉卿	33.33%
	41724057	郭真铃	33.33%
	41724059	曹璐然	33.33%

指导老师 ： 李建江

实验学时 ： 4 学时

时间 ： 2020 年 5 月 4 日

成绩

一、实验目的与要求

1. 掌握 Linpack 和 HPL 的相关知识。

2. 完成 HPL 的安装与配置。

3. 运行 HPL，测试计算机的性能。

（1）每组组内成员分别测试各自电脑性能并进行性能比较。

（2）有条件的小组，可将组内成员的电脑构建为小“集群”，测试该“集群”的性能。

4. 调整相关参数或优化程序代码，测试计算机的性能。与之前测得的计算机性能进行比较，并分析性能变化的原因。

（1）可使用 VTune 等工具对程序进行性能分析，找出其热点/瓶颈。

（2）可使用第三方工具，如：Intel Parallel Studio xe（学生可免费申请）、Intel 编译器、MKL 等。

二、实验环境

1. 硬件环境：计算机若干台（每小组组内成员的电脑）。

2. 软件环境：Linux、HPL、MPI、GCC、VTune、Intel Parallel Studio xe、Intel 编译器、MKL 等。

三、实验内容与步骤（请描述过程并进行截屏）

1. 安装 MPI

①在官网 http://www.mpich.org/downloads/下载 mpich 安装包：

mpich-3.3.2 (stable release)	MPICH	[http]	26 MB

②放到一个文件夹中并进行解压：

```
caolr@caolr-u:~/ep-mpi$ tar -zxf mpich-3.3.2.tar.gz
```

③解压后进入文件夹，并对配置文件指定安装路径：

```
caolr@caolr-u:~/ep-mpi/mpich-3.3.2$ ./configure -prefix=/home/caolr/mpi
```

④可能会遇到没有 g 编译器的情况，如下图：

```
th src/env/cc_shlib.conf)
configure: error: No Fortran 77 compiler found. If you don't need to
       build any Fortran programs, you can disable Fortran support using
       --disable-fortran. If you do want to build Fortran
       programs, you need to install a Fortran compiler such as gfortran
       or ifort before you can proceed.
```

安装 g77，首先更新源：

```
caolr@caolr-u:~/ep-mpi/mpich-3.3.2$ sudo gedit /etc/apt/sources.list
```

添加代码：

```
deb [trusted=yes] http://old-releases.ubuntu.com/ubuntu/ hardy universe
deb-src [trusted=yes] http://old-releases.ubuntu.com/ubuntu/ hardy universe
deb [trusted=yes] http://old-releases.ubuntu.com/ubuntu/ hardy-updates universe
deb-src [trusted=yes] http://old-releases.ubuntu.com/ubuntu/ hardy-updates universe
```

保存后关闭，然后下载 g77：

```
caolr@caolr-u:~$ sudo apt-get install gfortran
```

⑤回到解压目录再次配置安装路径，配置完成后，进行 make 和 make install：

```
config.status: executing depfiles commands
config.status: executing libtool commands
Configuration completed.
```

```
caolr@caolr-u:~/ep-mpi/mpich-3.3.2$ make
```

```
caolr@caolr-u:~/ep-mpi/mpich-3.3.2$ sudo make install
```

⑥要注意我们所需使用的命令（如 mpicc、mpirun）是我们新添加的，必须添加绝对路径才能正常使用。为了能全局使用，需要配置一下环境变量。同时要注意默认 shell 是 bash 配置还是 zsh 配置。打开 bachrc：

```
caolr@caolr-u:~/ep-mpi/mpich-3.3.2$ sudo getid ~/.bashrc
```

将 PATH 指定为安装路径中的 bin 目录：

```
export PATH=/home/caolr/mpi/bin:$PATH
```

保存，并更新配置文件：

```
caolr@caolr-u:~/ep-mpi/mpich-3.3.2$ source .bashrc
```

⑦至此配置完成，可以利用安装包中的 hellow 原文件进行测试观察是否配置成功：

```
caolr@caolr-u: ~/ep-mpi/mpich-3.3.2/examples
config.status  doc           Makefile.in  README.envvar
caolr@caolr-u:~/ep-mpi/mpich-3.3.2$ cd examples
caolr@caolr-u:~/ep-mpi/mpich-3.3.2/examples$ mpirun -n 1 ./hellow
Hello world from process 0 of 1
caolr@caolr-u:~/ep-mpi/mpich-3.3.2/examples$ mpirun -n 2 ./hellow
Hello world from process 0 of 2
Hello world from process 1 of 2
caolr@caolr-u:~/ep-mpi/mpich-3.3.2/examples$ mpirun -n 3 ./hellow
Hello world from process 0 of 3
Hello world from process 2 of 3
Hello world from process 1 of 3
caolr@caolr-u:~/ep-mpi/mpich-3.3.2/examples$ mpirun -n 4 ./hellow
Hello world from process 0 of 4
Hello world from process 3 of 4
Hello world from process 2 of 4
Hello world from process 1 of 4
caolr@caolr-u:~/ep-mpi/mpich-3.3.2/examples$ mpirun -n 5 ./hellow
Hello world from process 0 of 5
Hello world from process 4 of 5
Hello world from process 1 of 5
Hello world from process 3 of 5
Hello world from process 2 of 5
caolr@caolr-u:~/ep-mpi/mpich-3.3.2/examples$
caolr@caolr-u:~/ep-mpi/mpich-3.3.2/examples$
```

2. 配置 HPL

（1）安装 BLAS

①在官网 http://www.netlib.org/blas/index.html 下载 BLAS 安装包：

REFERENCE BLAS Version 3.8.0

- Download blas-3.8.0.tgz
- Updated November 2017
- Quick Reference Guide

②解压下载的压缩文件：

```
caolr@caolr-u:~/ep-blas$ cd blas-apt
caolr@caolr-u:~/ep-blas/blas-apt$ tar -zxf blas-3.8.0.tgz
```

③进入目录并进行编译和连接：

```
caolr@caolr-u:~/ep-blas/blas-apt$ cd BLAS-3.8.0
caolr@caolr-u:~/ep-blas/blas-apt/BLAS-3.8.0$ make
```

```
caolr@caolr-u:~/ep-blas/blas-apt/BLAS-3.8.0$ ar rv libblas.a *.o
```

（2）安装 CBLAS

①用命令下载 CBLAS 安装包：

```
caolr@caolr-u:~/ep-cblas$ wget http://www.netlib.org/blas/blast-forum/cblas.tgz
```

②解压压缩包：

```
caolr@caolr-u:~/ep-cblas$ tar -zxf cblas.tgz
```

③进入目录并将 blas_LINUX.a 复制到当前目录：

```
caolr@caolr-u:~/ep-cblas$ cd CBLAS
caolr@caolr-u:~/ep-cblas/CBLAS$ cp BLAS-3.8.0/blas_LINUX.a ./
```

④修改文件 Makefile.in 文件中的 BLLIB：

```
caolr@caolr-u:~/ep-cblas/CBLAS$ vim Makefile.in
```

修改为：

```
BLLIB = ../blas_LINUX.a
CBLIB = ../lib/cblas_$(PLAT).a
```

⑤进行编译：

```
caolr@caolr-u:~/ep-cblas/CBLAS$ make
```

⑥最后进行测试：

```
caolr@caolr-u: ~/ep-hpl/hpl-2.3
make[1]: Leaving directory '/home/caolr/ep-cblas/CBLAS/testing'
caolr@caolr-u:~/ep-cblas/CBLAS$ ./yesting/xzcblat1
bash: ./yesting/xzcblat1: 没有那个文件或目录
caolr@caolr-u:~/ep-cblas/CBLAS$ ./testing/xzcblat1
 Complex CBLAS Test Program Results


 Test of subprogram number  1          CBLAS_ZDOTC
                                    ----- PASS -----

 Test of subprogram number  2          CBLAS_ZDOTU
                                    ----- PASS -----

 Test of subprogram number  3          CBLAS_ZAXPY
                                    ----- PASS -----

 Test of subprogram number  4          CBLAS_ZCOPY
                                    ----- PASS -----

 Test of subprogram number  5          CBLAS_ZSWAP
                                    ----- PASS -----

 Test of subprogram number  6          CBLAS_DZNRM2
                                    ----- PASS -----

 Test of subprogram number  7          CBLAS_DZASUM
                                    ----- PASS -----

 Test of subprogram number  8          CBLAS_ZSCAL
                                    ----- PASS -----

 Test of subprogram number  9          CBLAS_ZDSCAL
                                    ----- PASS -----

 Test of subprogram number 10          CBLAS_IZAMAX
                                    ----- PASS -----
caolr@caolr-u:~/ep-cblas/CBLAS$ cd
```

（3）安装并配置 hpl

①下载 hpl 安装包 http://www.netlib.org/benchmark/hpl/index.html

```
--------------------------------------------------------------------------------

file    hpl-2.3.tar.gz
for     HPL 2.3 - A Portable Implementation of the High-Performance Linpack
,       Benchmark for Distributed-Memory Computers
by      Antoine Petitet, Clint Whaley, Jack Dongarra, Andy Cleary, Piotr Luszczek
Updated: December 2, 2018
```

②将 cblas_LINUX.a 和 blas_LINUX.a 复制到同一目录下，我这里将 blas_LINUX.a 复制到了 cblas 的 lib 目录下，因为这个目录存放着 cblas_LINUX.a

```
caolr@caolr-u:~$ cp /home/caolr/ep-blas/BLAS-3.8.0/blas_LINUX.a /home/caolr/ep-c
blas/CBLAS/lib
```

③将下载的 hpl 安装包解压

```
caolr@caolr-u:~/ep-hpl$ tar -zxf hpl-2.3.tar.gz
```

④进入解压目录并将 setup 目录中对应之前的 MPI 和 CBLAS 的 Make.Linux_PII_CBLAS 文件复制到解压目录，并重命名为 Make.ep。Make.Linux_PII_CBLAS 文件代表 Linux 操作系统，PII 平台，采用 CBLAS 库。这里的命名很重要，要记住它。

```
caolr@caolr-u:~/ep-hpl/hpl-2.3$ cp setup/Make.Linux_PII_CBLAS ./Make.ep
```

⑤然后修改刚才复制出来的文件。

```
caolr@caolr-u:~/ep-hpl/hpl-2.3$ vim Make.ep
```

arch 修改为刚刚命名的名字 Make.后面的部分

TOPdir 修改为 hpl 安装路径也是解压路径

MPdir 修改为 mpich 的安装路径（注意这里是安装路径曾经配置过的）

MPlib 修改为$(MPdir)/lib/libmpicxx.a

LAdir 修改为②中两个文件共存路径

LAlib 修改为$(LAdir)/cblas_LINUX.a $(LAdir)/blas_LINUX.a

CC 修改为 mpich 安装路径/bin/mpicc

LINKER 修改为 mpich 安装路径/bin/mpif77

```
# ----------------------------------------------------------------------
# - Platform identifier ------------------------------------------------
# ----------------------------------------------------------------------
#
ARCH         = ep
#
# ----------------------------------------------------------------------
# - HPL Directory Structure / HPL library ------------------------------
# ----------------------------------------------------------------------
#
TOPdir       = /home/caolr/ep-hpl/hpl-2.3
INCdir       = $(TOPdir)/include
BINdir       = $(TOPdir)/bin/$(ARCH)
LIBdir       = $(TOPdir)/lib/$(ARCH)
#
HPLlib       = $(LIBdir)/libhpl.a
#
# ----------------------------------------------------------------------
# - Message Passing library (MPI) --------------------------------------
```

```
# MPinc tells the  C  compiler where to find the Message Passing library
# header files,  MPlib  is defined  to be the name of  the library to be
# used. The variable MPdir is only used for defining MPinc and MPlib.
#
MPdir        = /home/caolr/mpi
MPinc        = -I$(MPdir)/include
MPlib        = $(MPdir)/lib/libmpicxx.a
#
# ----------------------------------------------------------------------
# - Linear Algebra library (BLAS or VSIPL) -----------------------------
# ----------------------------------------------------------------------
# LAinc tells the  C  compiler where to find the Linear Algebra  library
# header files,  LAlib  is defined  to be the name of  the library to be
# used. The variable LAdir is only used for defining LAinc and LAlib.
#
LAdir        = /home/caolr/ep-cblas/CBLAS/lib
LAinc        =
LAlib        = $(LAdir)/cblas_LINUX.a $(LAdir)/blas_LINUX.a
#
# ----------------------------------------------------------------------
# - F77 / C interface --------------------------------------------------
# ----------------------------------------------------------------------
# You can skip this section  if and only if  you are not planning to use
# a  BLAS  library featuring a Fortran 77 interface.  Otherwise,  it  is
```

```
#
CC           = /home/caolr/mpi/bin/mpicc
CCNOOPT      = $(HPL_DEFS)
CCFLAGS      = $(HPL_DEFS) -fomit-frame-pointer -O3 -funroll-loops
#
# On some platforms,  it is necessary  to use the Fortran linker to find
# the Fortran internals used in the BLAS library.
#
LINKER       = /home/caolr/mpi/bin/mpif77
LINKFLAGS    = $(CCFLAGS)
#
```

⑥进行编译，在 hpl/目录下执行 make arch=<arch>，<arch>即为 Make.<arch>文件的后缀，生成可执行文件 xhpl。

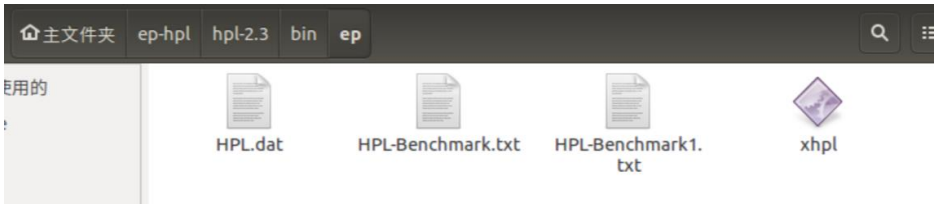
```
caolr@caolr-u:~/ep-hpl/hpl-2.3$ make arch=ep
```

在这里如果是集群测试，就将 LinPack 目录复制到集群中其余节点相同的目录下。

⑦进入 bin 目录中有个和刚才命名的名字同名的文件夹，此时可以运行了。

```
caolr@caolr-u:~/ep-hpl/hpl-2.3$ cd bin/ep
caolr@caolr-u:~/ep-hpl/hpl-2.3/bin/ep$ mpirun -np 4 ./xhpl > HPL-Benchmark.txt
```

没有报错即运行成功。运行结果保存在当前这个路径下，与代码中同名的 txt 中，查看即可。



⑧注意：在复制 Make.Linux_PII_FBLAS 到上层目录重命名时，对于命名规则和 cpu 架构有关，可以通过命令 cat /proc/cpuinfo 来查看，否则会发生如下错误：

```
/usr/bin/ld: cannot open output file /usr/local/HPL/hpl-2.3/bin/jyq/xhpl: Permis
sion denied
collect2: error: ld returned 1 exit status
Makefile:76: recipe for target 'dexe.grd' failed
make[2]: *** [dexe.grd] Error 1
make[2]: Leaving directory '/usr/local/HPL/hpl-2.3/testing/ptest/jyq'
Make.top:64: recipe for target 'build_tst' failed
make[1]: *** [build_tst] Error 2
make[1]: Leaving directory '/usr/local/HPL/hpl-2.3'
Makefile:72: recipe for target 'build' failed
make: *** [build] Error 2
```

3. 运行并进行测试

使用 lshw、lscpu 等命令查看各自电脑配置，结果如下。

Item	**Configuration**
Computer1（金玉卿）	CPU: Intel Core i5-2410M*4, 2.30GHz
	Memory: 9.6G
	Hard disk: 491.2G
	Cacheline（L2）：256K
Computer2（郭真铃）	CPU:Intel Core i7-8550U CPU*8 @ 1.80GHz
	Memory: 7826MB（约 8G）
	Hard disk: 80G
	Cacheline（L2）：256K
Computer3（曹璐然）	CPU: Pentium® Dual-Core CPU E5800 @3,2GHz *2
	Memory: 1992M
	Hard Disk: 465G
	Cacheline(L2): 2048K

Item	**Description**	**Version**

OS	Ubuntu	18.04
Compiler	GCC	7.5.0
MPI	MPI	MPICH-3.3.2
BLAS	BLAS	BLAS-3.8.0
HPL	HPL	HPL-2.3

运行时进入 bin 目录中生成的项目文件夹，执行 mpirun -np x ./xhpl > [文件名]命令，此处 x 表示线程数，若小于某一组 PQ 的值会报错；文件名可自定义方便之后的浏览。生成文件保存在项目文件夹中。

4. 优化（详细阐述如何进行优化）并进行测试

4.1 HPL.dat 中参数的优化

HPL.dat 文件内容如下：

```
HPLinpack benchmark input file                                  #1
Innovative Computing Laboratory, University of Tennessee        #2
HPL.out      output file name (if any)                          #3
6            device out (6=stdout,7=stderr,file)                #4
1            # of problems sizes (N)                            #5
5000         Ns                                                 #6
1            # of NBs                                           #7
128          NBs                                                #8
0            PMAP process mapping (0=Row-,1=Column-major)       #9
3            # of process grids (P x Q)                         #10
2 1 4        Ps                                                 #11
2 4 1        Qs                                                 #12
16.0         threshold                                          #13
3            # of panel fact                                    #14
0 1 2        PFACTs (0=left, 1=Crout, 2=Right)                  #15
2            # of recursive stopping criterium                  #16
2 4          NBMINs (>= 1)                                      #17
1            # of panels in recursion                           #18
2            NDIVs                                              #19
3            # of recursive panel fact.                         #20
0 1 2        RFACTs (0=left, 1=Crout, 2=Right)                  #21
1            # of broadcast                                     #22
0            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)       #23
1            # of lookahead depth                               #24
0            DEPTHs (>=0)                                       #25
2            SWAP (0=bin-exch,1=long,2=mix)                     #26
64           swapping threshold                                 #27
0            L1 in (0=transposed,1=no-transposed) form          #28
0            U  in (0=transposed,1=no-transposed) form          #29
```

1	Equilibration (0=no,1=yes)	#30
8	memory alignment in double (> 0)	#31

下面逐个简要说明每个参数的含义及一般配置：

（1）第 1、2 行为注释说明行，不需要作修改

（2）第 3、4 行说明输出结果文件的形式

“device out”为“6”时，测试结果输出至标准输出(stdout)

“device out”为“7”时，测试结果输出至标准错误输出(stderr)

“device out”为其他值时，测试结果输出至第三行所指定的文件中

（3）第 5，6 行说明求解矩阵的次数和大小 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。因此，对于一般系统而言，要尽量增大矩阵规模 N 的同时，又要保证不使用系统缓存。因为操作系统本身需要占用一定的内存，除了矩阵(N×N)之外，HPL 还有其他的内存开销，另外通信也需要占用一些缓存。矩阵占用系统总内存的 80%左右为最佳，即 N×N×8=系统总内存×80%。

（4）第 7、8 行说明求解矩阵分块的大小 NB

为提高数据的局部性，从而提高整体性能，HPL 采用分块矩阵的算法。分块的大小对性能有很大的影响，NB 的选择和软硬件许多因素密切相关。

下面给出一些平台上的 NB 选择。

平台	L2 Cache	运行模式	数学库	NB
Intel P4 Xeon	512KB	32 位	ATLAS	400
			MKL	384
			GOTO	192
Intel P4 Xeon (Nocona)	1MB	64 位	GOTO	96
AMD Opteron	1MB	64 位	GOTO	232

NB 值的选择主要是通过实际测试得到最优值。但 NB 的选择上还是有一些规律可寻，如:NB 不可能太大或太小，一般在 256 以下:NB×8 一定是 Cache line 的倍数等。

另外，NB 大小的选择还跟通信方式、矩阵规模、网络、处理器速度等有关系。一般通过单节点或单 CPU 测试可以得到几个较好的 NB 值，但当系统规模增加、问题规模变大，有些 NB 取值所得性能会下降。所以最好在小规模测试时选择 3 个左右性能不错的 NB，再通过大规模测试检验这些选择。

（5）第 9 行是选择处理器阵列是按列的排列方式还是按行的排列方式。

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。

（6）第 10-12 行说明二位处理器网络（P×Q），二位处理器网格（P×Q）的有以下几

个要求：

P×Q=进程数，这是 HPL 的硬性规定；

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。

当 Q/4≤P≤Q 时，性能最优。

P≤Q：一般来说，P 的值尽量取得小一点，因为列项通信量要远大于横向通信。

$P=2^n$，即 P 最好选择 2 的幂次，HPL 中，L 分解的列项通信采用二元交换法，当列项处理器个数 P 为 2 的幂时，性能最优。

（7）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。

（8）第 14-21 行指明 L 分解的方式。

在 HPL 官方文档中，推荐的设置为：

```
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)
```

（9）第 22、23 行说明 L 的横向广播方式。

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。

推荐的配置为：

```
2            # of broadcast
1 3          BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
```

（10）第 24、25 行说明横向通信的通信深度，依赖于机器的配置和问题规模的大小。

推荐配置为：

```
2            # of lookahead depth
0 1          DEPTHs (>=0)
```

（11）第 26、27 行说明 U 的广播算法。

推荐配置为：

```
2            SWAP (0=bin-exch,1=long,2=mix)
60           swapping threshold
```

（12）第 28、29 行分别说明 L 和 U 的数据存放格式。

推荐配置为：

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

（13）第 30 行主要在回代中使用，一般使用其默认值。

（14）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其地址，处于安全考虑，可以选择 8。

4.2 其他性能优化

①MPI

对于常用的 MPICH 来说，安装编译 MPICH 时，使其节点内采用共享内存进行通信可以提升一部分性能，在 configure 时，设置”—with-comm=shared”。

对于 GM 来说，在找到路由以后，将每个节点的 gm_mapper 进程 kill 掉，大概有一个百分点的性能提高。

②操作系统

操作系统层的性能优化方法，如裁剪内核、改变页面大小、调整内核参数、调整网络参数等等。

四、实验结果与分析

Computer1（金玉卿）：

1. 不进行优化，分析在什么情况下（N、NB、P、Q 等），可以获得更好的性能。

在未进行优化的前提下，只修改进程数和 PQ 的值。进程数分别取 1、2、4、8、16，二位处理器网格（P×Q）的要求如下：

P×Q=进程数；

P≤Q；一般来说，P 的值尽量取得小一点，因为列项通信量要远大于横向通信。

$P=2^n$，即 P 最好选择 2 的幂次，HPL 中，L 分解的列项通信采用二元交换法，当列项处理器个数 P 为 2 的幂时，性能最优。因此可以设计进程数对应的 P 和 Q 的取值如下表：

进程数	P	Q
1	1	1
2	1	2
4	1	4
	2	2
8	1	8
	2	4
16	1	16
	2	8
	4	4

N 为默认的 29、30、34、35，NB 为默认的 1、2、3、4，在不同线程时，得到的最佳 Gflops 如下：

进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）

1	35	4	1	1	0.00	6.8213e-01
2	35	4	1	2	0.00	3.3228e-01
4	35	4	1	4	0.00	3.0969e-01
8	35	4	1	8	0.10	3.0428e-04
16	35	4	1	16	0.17	1.6630e-04

从测试结果表中可以看出，在 N 的值为 29、30、34、35 的情况下，一直是 N=35 获得最优值，从目前结果推测 N 似乎越大越好。电脑配置 cpu 核数为 4，在进程数超过电脑核数时，运行效果明显变差，在 16 进程下花费的时间较久，且测得的实际峰值速度最低，综上所述，在 N=35,NB=4,P=1,Q=1 时取得最好的性能，实际峰值速度达到 0.58213Gflops

2. 展示优化后的测试结果，并详细分析调整相关参数或优化程序代码后能够获得更高的性能的原因。

从未优化时的测试数据可以推测 N 和 NB 的值似乎越大越好，同时进程数尽量不超过 cpu 的核数，即一个进程对应一个 CPU 获得的性能较好。

对于 NB 的取值，主要是通过实际测试得到最优值，但由经验可知，NB 不能太大或太小，一般在 256 以下；NB×8=cacheline 的倍数，通过单 CPU 测试可以得到较好的 NB 值，所以在这里取 N=200，P=1，Q=1，由于 cacheline=256K，所以预取 NB 的值为 32，64，96，128，160，192，进行测试，测试结果如下表：

进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）
1	200	32	1	1	0.00	2.8736e+00
1	200	64	1	1	0.00	2.6701e+00
1	200	96	1	1	0.00	2.5779e+00
1	200	128	1	1	0.00	2.4856e+00
1	200	192	1	1	0.00	2.4256e+00
1	200	256	1	1	0.00	2.4812e+00

由上表可知，NB 取 32，64，96 时，性能较好。

对于 N 的取值，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。因此，对于一般系统而言，要尽量增大矩阵规模 N 的同时，又要保证不使用系统缓存。因为操作系统本身需要占用一定的内存，除矩阵（N×N）外，HPL 有其他的内存开销，同时通信也会占用一些缓存。矩阵占用系统总内存的 80%左右为最佳，即 N×N×8=系统内存（byte）×80%。

由内存大小 9.6G 大概计算出 N 的大小为 30983，此时固定 P=1、Q=4 或 P=1、Q=2，NB=64，调整 N 的大小，对 N 取值为 2000，3000，4000，5000，6000，进行测试，测试结果如下表：

进程数	N	NB	P	Q	执行时间（s）	Gflops
4	2000	64	1	4	1.39	3.8346e+00
4	3000	64	1	4	6.45	2.7909e+00
4	4000	64	1	4	17.29	2.4686e+00
4	5000	64	1	4	35.52	2.3469e+00
4	6000	64	1	4	61.89	2.3276e+00
4	1500	64	1	4	1.54	1.4646e+00
4	1200	64	1	4	0.93	1.2443e+00
4	1800	64	1	4	1.04	3.7286e+00
4	2200	64	1	4	2.22	3.2013e+00
4	2300	64	1	4	2.57	3.1584e+00

从上述表格可知，在 N 为 2000 左右时实际峰值速度较大，综合测试表格如下：

进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）
2	1900	32	1	2	0.77	6.1719e+00
2	2000	32	1	2	0.89	6.5645e+00
2	2100	32	1	2	0.89	6.1799e+00
2	1900	64	1	2	0.79	5.9815e+00
2	2000	64	1	2	0.93	5.8183e+00
2	2100	64	1	2	1.09	5.7958e+00
4	1950	32	1	4	0.84	5.9880e+00
4	2000	32	1	4	0.91	5.9802e+00
4	2050	32	1	4	1.00	5.8315e+00
4	2100	32	1	4	1.05	5.9727e+00

由上表可知，当 N=2000，NB=32，P=1，Q=2 时，实际峰值速度最大为 6.5645Gflops。与优化前的最好结果 0.68213Gflops 比较，性能提高了 n=9.62 倍。

Computer2（郭真铃）：

1. 不进行优化，分析在什么情况下（N、NB、P、Q 等），可以获得更好的性能。

进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）
1	35	4	1	1	0.00	9.4514e-01
2	35	4	1	2	0.00	5.4761e-01
4	35	4	1	4	0.00	8.5634e-01
8	35	4	1	8	0.00	2.6093e-01
16	35	4	1	16	0.13	2.3769e-04

以上的结果是从各种进程数中挑选 Gflops 最大的结果。各个测试结果除去 P、Q 不同外，N、NB 的值的设置是一样的，都是 N：29、30、34、35；NB：1、2、3、4；而 P、Q 对应进程数，分别设置如下（表 2）：

进程数	P	Q
1	1	1
2	1	2
4	1	4
	2	2
8	1	8
	2	4
16	1	16
	2	8
	4	4

从测试结果表中可以看出，在 N 分别为 29、30、34、35 的情况下，一直是 N=35 获得最优值，从目前结果可见 N 似乎越大越好。在 NB 分别为 1、2、3、4 的情况下，一直是 NB=4 获得最优值，从目前结果可见 NB 似乎越大越好。结合测试结果表和表 2，一直是 P 为 1 的情况最优。总而言之，在未优化时，设定 N=35、NB=4、P=1、Q=1，有结果 Gflops =9.4514e-01。

2. 展示优化后的测试结果，并详细分析调整相关参数或优化程序代码后能够获得更高的性能的原因。

从上面的结果中可知进程数为 1、4 的结果相对较优，因此选取进程数分别为 1、4 两种情况来进行测试。

其次确定 N 值，根据他人经验可知，有 N×N×8=系统内存（以 Byte 为单位）×80%，根据我的硬件情况得到 N 要小于等于 28646，此时固定 P=1、Q=1、NB=4，调整 N，最终发现 N=5000 有较优值。

然后设定 N=5000，使 NB 分别为 4、8、16、32、64、128；P*Q 分别为 1、4，结果如下：

进程数	N	NB	P	Q	执行时间（s）	Gflops
1	5000	4	1	1	30.48	2.7349e+00
1	5000	8	1	1	30.47	2.7363e+00
1	5000	16	1	1	30.46	2.7370e+00
1	5000	32	1	1	28.82	2.8931e+00
1	5000	64	1	1	26.07	3.1976e+00
1	5000	128	1	1	23.11	3.6081e+00
4	5000	4	1	4	11.82	7.0536e+00
4	5000	8	1	4	7.99	1.0433e+01
4	5000	16	1	4	7.72	1.0793e+01
4	5000	32	1	4	7.66	1.0890e+01
4	5000	64	1	4	8.64	9.6540e+00
4	5000	128	1	4	12.32	6.7698e+00
4	5000	4	2	2	11.94	6.9823e+00
4	5000	8	2	2	7.39	1.1285e+01
4	5000	16	2	2	7.17	1.1631e+01
4	5000	32	2	2	7.35	1.1344e+01
4	5000	64	2	2	7.44	1.1204e+01
4	5000	128	2	2	8.59	9.6999e+00

从上述表格可知，设定 N=5000、NB=16、P=2、Q=2 时，可得到最优结果 Gflops=1.1631e+01，相较于未优化前 Gflops =9.4514e-01，性能提高了 n=12.3 倍。

Computer3（曹璐然）：

1.不进行优化，分析在什么情况下（N、NB、P、Q 等），可以获得更好的性能。

首先在未进行优化的前提下，只是修改进程数和 PQ 的值。进程数分别取 1、2、4、8、16 时，P、Q 根据分析中的规则取值为：

进程数	P	Q
1	1	1
2	1	2
4	1	4
	2	2
8	1	8
	2	4
16	1	16
	2	8
	4	4

N 为默认的 29、30、34、35，NB 为默认的 1、2、3、4。不同线程时，得到的最佳 Gflops 如下：

进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）
1	35	4	1	1	0	9.8149E-01
2	35	4	1	2	0	4.0378E-01
4	35	3	1	4	0.21	5.9397E-04
8	35	4	1	8	0.12	2.5354E-04
16	35	4	1	16	0.51	8.8997E-05

通过上表再结合我的 cpu 是两核可知，当进程数超过 cpu 核数的时候效果明显变差。甚至我的 16 线程根本跑不完。所以在线程为 1、2 时表现较好。此时的 P、Q 相乘应为线程数，且 P 为较小值，可以得到较好的 Gflops 值。

2.展示优化后的测试结果，并详细分析调整相关参数或优化程序代码后能够获得更高的性能的原因。

结合我的设备 cpu 核数为 2，所以 P、Q 取 1、1 和 1、2。根据未优化所得结果可以看出有 N、NB 更大时会有更好的结果的趋势。所以接下来寻找 N、NB 的最大值。

首先固定其他数据，测试一些 NB 的值，找出比较合适 NB。矩阵分块的大小对计算的影响较大，分块过大容易造成过载不平衡；分块过小通讯开销会很大同样影响性能。我这里根据所谓的 NB*8=Cacheline 的倍数这个经验，测试了一些数据结果如下：

进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）
2	50	64	1	2	0	1.0676

2	50	96	1	2	0	1.0806
2	50	128	1	2	0	1.0556
2	50	160	1	2	0	1.0496
2	50	192	1	2	0	1.0496
2	50	256	1	2	0	1.0466

可以看出 NB 取 64、96、128 时，表现较好，保留这三个值带入到后面的测试。

最后寻找 N，因为 N 较大所以放在后面比较节省测试的时间。N 越大时，有效计算所占的比例也越大，系统浮点处理性能也就越高；但是过大之后内存不够用，会发生数据交换，磁盘交换区的性能远低于内存，反而降低了有效计算的比例。经验公式指出 N 的理论巅峰值为 N*N*8=内存*80%，根据此公式可以算出 N≈15764。但是发现在实际实验中取不到这么大，下面是对 N 进行一些取值发现的比较接近最大值的情况：

进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）
1	2000	128	1	1	4.25	1.2566
1	3000	128	1	1	19.52	9.2322E-01
2	2000	128	1	2	3.24	1.6464
2	3000	128	1	2	13.60	1.3244
2	1500	128	1	2	1.18	1.9019
2	500	128	1	2	0.03	2.9724
2	700	128	1	2	0.08	3.0440
2	800	128	1	2	0.11	3.2292
2	850	128	1	2	0.14	3.0397
2	900	128	1	2	0.15	3.2070
2	950	128	1	2	0.22	2.6619
2	1000	128	1	2	0.23	2.9597

由上表可以看出 N 在 800 左右且线程等于 cpu 核数时 Gflops 比较高，下面综合各项数据再进行一次测试，其中得到的一些较好结果如下表：

进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）
2	750	64	1	2	0	3.6489
2	750	96	1	2	0.08	3.5078

2	760	64	1	2	0.08	3.6770
2	780	96	1	2	0.09	3.4295
2	800	64	1	2	0.10	3.7663
2	820	64	1	2	0.12	3.7282
2	840	32	1	2	0.11	3.6405
2	840	64	1	2	0.10	3.7885
2	870	64	1	2	0.12	3.7308
2	900	64	1	2	0.13	3.7686

由上表可知，当 N=840，NB=64，P=1，Q=2 时，Gflops 达到巅峰为 3.7885。与优化前的最好结果 4.0378E-01 比较，性能提高了 n=9.38 倍。

汇总：

我们小组的各台电脑测试结果及其优化结果如下所示：

	进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）	优化倍数
computer1								
改进前	1	35	4	1	1	0.00	6.8213e-01	9.62
改进后	2	2000	32	1	2	0.89	6.5645e+00	
computer2								
改进前	1	35	4	1	1	0.00	9.4514e-01	12.3
改进后	4	5000	16	2	2	7.17	1.1631e+01	
computer3								
改进前	1	35	4	1	1	0	9.8149E-01	9.38
改进后	2	840	64	1	2	0.10	3.7885	

五、实验心得体会

通过本次实验学习掌握了 Linpack 基准测试程序，该测试程序采用高斯消元法求解一元 N 次稠密线性代数方程组，对高性能计算机进行测试，同时 HPL 可以在用户不修改任意测试程序的基础上，调节问题规模大小，使用 CPU 的数目来执行此测试程序，为了完成实验，在安装 HPL 之前，在 Linux 系统中安装编译器、并行环境 MPI 以及基本线性代数子方程（BLAS），其中并行环境选择了 MPICH，HPL 的运行采用 BLAS 库与 CBLAS 库，并对其进行配置，基本掌握了 Linpack 测试的方法和步骤。

在整个试验过程中首先掌握了在已有 win 的情况下安装 Linux 双系统的方法，其中遇到硬盘模式不兼容的问题，在操作系统方面了解了更多的知识。其次从零开始学习 Linpack 测试的方法的过程中，通过阅读大量文章了解测试相关知识，综合各类文章总结了安装过程与配置过程，对学习能力有很大的提高。在优化过程中掌握了 HPL 测试中参数的含义，并在对其修改中学习了根据计算机配置寻找最佳参数的规律。

六、参考文献

1. https://wenku.baidu.com/view/c4e3447a168884868762d6fb.html

2. https://blog.csdn.net/sishuiliunian0710/article/details/20493101/

3. https://wenku.baidu.com/view/ec7829290066f5335a8121a2.html

附 1：

北京科技大学实验报告

学院： 计通学院 专业：计算机科学与技术 班级：计 172

姓名：金玉卿 郭真铃 曹璐然 学号：41724051 41724057 41724059

实验日期：2020 年 5 月 14 日

实验名称：

Linpack 性能测试与优化

实验目的：

1. 掌握 Linpack 和 HPL 的相关知识。

2. 完成 HPL 的安装与配置。

3. 运行 HPL，测试计算机的性能。

（1）每组组内成员分别测试各自电脑性能并进行性能比较。

（2）有条件的小组，可将组内成员的电脑构建为小“集群”，测试该“集群”的性能。

4. 调整相关参数或优化程序代码，测试计算机的性能。与之前测得的计算机性能进行比较，并分析性能变化的原因。

（1）可使用 VTune 等工具对程序进行性能分析，找出其热点/瓶颈。

（2）可使用第三方工具，如：Intel Parallel Studio xe（学生可免费申请）、Intel 编译器、MKL 等。

实验仪器：

1. 硬件环境：计算机若干台（每小组组内成员的电脑）。

2. 软件环境：Linux、HPL、MPI、GCC、VTune、Intel Parallel Studio xe、Intel 编译器、MKL 等。

实验原理：

LINPACK 是线性系统软件包(Linear system package) 的缩写。

Linpack 现在在国际上已经成为最流行的用于测试高性能计算机系统浮点性能的 benchmark。通过利用高性能计算机，用高斯消元法求解一元 N 次稠密线性代数方程组的测试，评价高性能计算机的浮点性能。

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，也叫高度并行计算基准测试，它对数组大小 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。前两种测试运行规模较小，已不是很适合现代计算机的发展，因此现在使用较多的测试标准为 HPL，而且阶次 N 也是 linpack 测试必须指明的参数。

LINPACK 压力测试的目的主要为检测系统中 CPU 的工作的稳定性及内存访问的稳定性。

实验内容与步骤：

1. 安装 MPI

①在官网 http://www.mpich.org/downloads/下载 mpich 安装包：

mpich-3.3.2 (stable release)	MPICH	[http]	26 MB

②放到一个文件夹中并进行解压：

```
caolr@caolr-u:~/ep-mpi$ tar -zxf mpich-3.3.2.tar.gz
```

③解压后进入文件夹，并对配置文件指定安装路径：

```
caolr@caolr-u:~/ep-mpi/mpich-3.3.2$ ./configure -prefix=/home/caolr/mpi
```

④可能会遇到没有 g 编译器的情况，如下图：

```
(in src/env/cc_shlib.conf)
configure: error: No Fortran 77 compiler found. If you don't need to
       build any Fortran programs, you can disable Fortran support using
       --disable-fortran. If you do want to build Fortran
       programs, you need to install a Fortran compiler such as gfortran
       or ifort before you can proceed.
```

安装 g77，首先更新源：

```
caolr@caolr-u:~/ep-mpi/mpich-3.3.2$ sudo gedit /etc/apt/sources.list
```

添加代码：

```
deb [trusted=yes] http://old-releases.ubuntu.com/ubuntu/ hardy universe
deb-src [trusted=yes] http://old-releases.ubuntu.com/ubuntu/ hardy universe
deb [trusted=yes] http://old-releases.ubuntu.com/ubuntu/ hardy-updates universe
deb-src [trusted=yes] http://old-releases.ubuntu.com/ubuntu/ hardy-updates universe
```

保存后关闭，然后下载 g77：

```
caolr@caolr-u:~$ sudo apt-get install gfortran
```

⑤回到解压目录再次配置安装路径，配置完成后，进行 make 和 make install：

```
config.status: executing depfiles commands
config.status: executing libtool commands
Configuration completed.
```

```
caolr@caolr-u:~/ep-mpi/mpich-3.3.2$ make
```

```
caolr@caolr-u:~/ep-mpi/mpich-3.3.2$ sudo make install
```

⑥要注意我们所需使用的命令（如 mpicc、mpirun）是我们新添加的，必须添加绝对路径才能正常使用。为了能全局使用，需要配置一下环境变量。同时要注意默认 shell 是 bash 配置还是 zsh 配置。打开 bachrc：

```
caolr@caolr-u:~/ep-mpi/mpich-3.3.2$ sudo getid ~/.bashrc
```

将 PATH 指定为安装路径中的 bin 目录：

```
export PATH=/home/caolr/mpi/bin:$PATH
```

保存，并更新配置文件：

```
caolr@caolr-u:~/ep-mpi/mpich-3.3.2$ source .bashrc
```

⑦至此配置完成，可以利用安装包中的 hellow 原文件进行测试观察是否配置成功：

```
caolr@caolr-u: ~/ep-mpi/mpich-3.3.2/examples
config.status  doc           Makefile.in  README.envvar
caolr@caolr-u:~/ep-mpi/mpich-3.3.2$ cd examples
caolr@caolr-u:~/ep-mpi/mpich-3.3.2/examples$ mpirun -n 1 ./hellow
Hello world from process 0 of 1
caolr@caolr-u:~/ep-mpi/mpich-3.3.2/examples$ mpirun -n 2 ./hellow
Hello world from process 0 of 2
Hello world from process 1 of 2
caolr@caolr-u:~/ep-mpi/mpich-3.3.2/examples$ mpirun -n 3 ./hellow
Hello world from process 0 of 3
Hello world from process 2 of 3
Hello world from process 1 of 3
caolr@caolr-u:~/ep-mpi/mpich-3.3.2/examples$ mpirun -n 4 ./hellow
Hello world from process 0 of 4
Hello world from process 3 of 4
Hello world from process 2 of 4
Hello world from process 1 of 4
caolr@caolr-u:~/ep-mpi/mpich-3.3.2/examples$ mpirun -n 5 ./hellow
Hello world from process 0 of 5
Hello world from process 4 of 5
Hello world from process 1 of 5
Hello world from process 3 of 5
Hello world from process 2 of 5
caolr@caolr-u:~/ep-mpi/mpich-3.3.2/examples$
caolr@caolr-u:~/ep-mpi/mpich-3.3.2/examples$
```

2. 配置 HPL

（1）安装 BLAS

①在官网 http://www.netlib.org/blas/index.html 下载 BLAS 安装包：

REFERENCE BLAS Version 3.8.0

- Download blas-3.8.0.tgz
- Updated November 2017
- Quick Reference Guide

②解压下载的压缩文件：

```
caolr@caolr-u:~/ep-blas$ cd blas-apt
caolr@caolr-u:~/ep-blas/blas-apt$ tar -zxf blas-3.8.0.tgz
```

③进入目录并进行编译和连接：

```
caolr@caolr-u:~/ep-blas/blas-apt$ cd BLAS-3.8.0
caolr@caolr-u:~/ep-blas/blas-apt/BLAS-3.8.0$ make
```

```
caolr@caolr-u:~/ep-blas/blas-apt/BLAS-3.8.0$ ar rv libblas.a *.o
```

（2）安装 CBLAS

①用命令下载 CBLAS 安装包：

```
caolr@caolr-u:~/ep-cblas$ wget http://www.netlib.org/blas/blast-forum/cblas.tgz
```

②解压压缩包：

```
caolr@caolr-u:~/ep-cblas$ tar -zxf cblas.tgz
```

③进入目录并将 blas_LINUX.a 复制到当前目录：

```
caolr@caolr-u:~/ep-cblas$ cd CBLAS
caolr@caolr-u:~/ep-cblas/CBLAS$ cp BLAS-3.8.0/blas_LINUX.a ./
```

④修改文件 Makefile.in 文件中的 BLLIB：

```
caolr@caolr-u:~/ep-cblas/CBLAS$ vim Makefile.in
```

修改为：

```
BLLIB = ../blas_LINUX.a
CBLIB = ../lib/cblas_$(PLAT).a
```

⑤进行编译：

```
caolr@caolr-u:~/ep-cblas/CBLAS$ make
```

⑥最后进行测试：

```
caolr@caolr-u: ~/ep-hpl/hpl-2.3
make[1]: Leaving directory '/home/caolr/ep-cblas/CBLAS/testing'
caolr@caolr-u:~/ep-cblas/CBLAS$ ./yesting/xzcblat1
bash: ./yesting/xzcblat1: 没有那个文件或目录
caolr@caolr-u:~/ep-cblas/CBLAS$ ./testing/xzcblat1
 Complex CBLAS Test Program Results


 Test of subprogram number  1          CBLAS_ZDOTC
                                    ----- PASS -----

 Test of subprogram number  2          CBLAS_ZDOTU
                                    ----- PASS -----

 Test of subprogram number  3          CBLAS_ZAXPY
                                    ----- PASS -----

 Test of subprogram number  4          CBLAS_ZCOPY
                                    ----- PASS -----

 Test of subprogram number  5          CBLAS_ZSWAP
                                    ----- PASS -----

 Test of subprogram number  6          CBLAS_DZNRM2
                                    ----- PASS -----

 Test of subprogram number  7          CBLAS_DZASUM
                                    ----- PASS -----

 Test of subprogram number  8          CBLAS_ZSCAL
                                    ----- PASS -----

 Test of subprogram number  9          CBLAS_ZDSCAL
                                    ----- PASS -----

 Test of subprogram number 10          CBLAS_IZAMAX
                                    ----- PASS -----
caolr@caolr-u:~/ep-cblas/CBLAS$ cd
```

（3）安装并配置 hpl

①下载 hpl 安装包 http://www.netlib.org/benchmark/hpl/index.html

```
##########################################################################

file    hpl-2.3.tar.gz
for     HPL 2.3 - A Portable Implementation of the High-Performance Linpack
,       Benchmark for Distributed-Memory Computers
by      Antoine Petitet, Clint Whaley, Jack Dongarra, Andy Cleary, Piotr Luszczek
Updated: December 2, 2018
```

②将 cblas_LINUX.a 和 blas_LINUX.a 复制到同一目录下，我这里将 blas_LINUX.a 复制到了 cblas 的 lib 目录下，因为这个目录存放着 cblas_LINUX.a

```
caolr@caolr-u:~$ cp /home/caolr/ep-blas/BLAS-3.8.0/blas_LINUX.a /home/caolr/ep-c
blas/CBLAS/lib
```

③将下载的 hpl 安装包解压

```
caolr@caolr-u:~/ep-hpl$ tar -zxf hpl-2.3.tar.gz
```

④进入解压目录并将 setup 目录中对应之前的 MPI 和 CBLAS 的 Make.Linux_PII_CBLAS 文件复制到解压目录，并重命名为 Make.ep。Make.Linux_PII_CBLAS 文件代表 Linux 操作系统，PII 平台，采用 CBLAS 库。这里的命名很重要，要记住它。

```
caolr@caolr-u:~/ep-hpl/hpl-2.3$ cp setup/Make.Linux_PII_CBLAS ./Make.ep
```

⑤然后修改刚才复制出来的文件。

```
caolr@caolr-u:~/ep-hpl/hpl-2.3$ vim Make.ep
```

arch 修改为刚刚命名的名字 Make.后面的部分

TOPdir 修改为 hpl 安装路径也是解压路径

MPdir 修改为 mpich 的安装路径（注意这里是安装路径曾经配置过的）

MPlib 修改为$(MPdir)/lib/libmpicxx.a

LAdir 修改为②中两个文件共存路径

LAlib 修改为$(LAdir)/cblas_LINUX.a $(LAdir)/blas_LINUX.a

CC 修改为 mpich 安装路径/bin/mpicc

LINKER 修改为 mpich 安装路径/bin/mpif77

```
# ----------------------------------------------------------------------
# - Platform identifier ------------------------------------------------
# ----------------------------------------------------------------------
#
ARCH         = ep
#
# ----------------------------------------------------------------------
# - HPL Directory Structure / HPL library ------------------------------
# ----------------------------------------------------------------------
#
TOPdir       = /home/caolr/ep-hpl/hpl-2.3
INCdir       = $(TOPdir)/include
BINdir       = $(TOPdir)/bin/$(ARCH)
LIBdir       = $(TOPdir)/lib/$(ARCH)
#
HPLlib       = $(LIBdir)/libhpl.a
#
# ----------------------------------------------------------------------
# - Message Passing library (MPI) --------------------------------------
```

```
# MPinc tells the  C  compiler where to find the Message Passing library
# header files,  MPlib  is defined  to be the name of  the library to be
# used. The variable MPdir is only used for defining MPinc and MPlib.
#
MPdir        = /home/caolr/mpi
MPinc        = -I$(MPdir)/include
MPlib        = $(MPdir)/lib/libmpicxx.a
#
# ----------------------------------------------------------------------
# - Linear Algebra library (BLAS or VSIPL) -----------------------------
# ----------------------------------------------------------------------
# LAinc tells the  C  compiler where to find the Linear Algebra  library
# header files,  LAlib  is defined  to be the name of  the library to be
# used. The variable LAdir is only used for defining LAinc and LAlib.
#
LAdir        = /home/caolr/ep-cblas/CBLAS/lib
LAinc        =
LAlib        = $(LAdir)/cblas_LINUX.a $(LAdir)/blas_LINUX.a
#
# ----------------------------------------------------------------------
# - F77 / C interface --------------------------------------------------
# ----------------------------------------------------------------------
# You can skip this section  if and only if  you are not planning to use
# a  BLAS  library featuring a Fortran 77 interface.  Otherwise,  it  is
```

```
#
CC           = /home/caolr/mpi/bin/mpicc
CCNOOPT      = $(HPL_DEFS)
CCFLAGS      = $(HPL_DEFS) -fomit-frame-pointer -O3 -funroll-loops
#
# On some platforms,  it is necessary  to use the Fortran linker to find
# the Fortran internals used in the BLAS library.
#
LINKER       = /home/caolr/mpi/bin/mpif77
LINKFLAGS    = $(CCFLAGS)
#
```

⑥进行编译，在 hpl/目录下执行 make arch=<arch>，<arch>即为 Make.<arch>文件的后缀，生成可执行文件 xhpl。

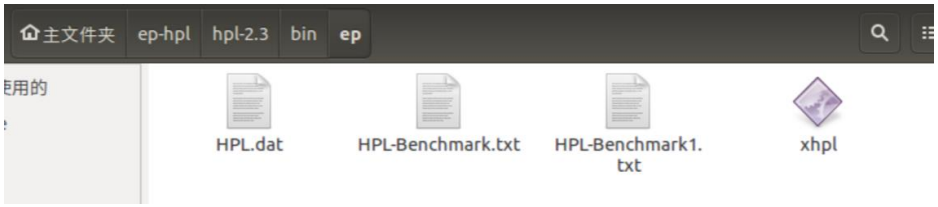
```
caolr@caolr-u:~/ep-hpl/hpl-2.3$ make arch=ep
```

在这里如果是集群测试，就将 LinPack 目录复制到集群中其余节点相同的目录下。

⑦进入 bin 目录中有个和刚才命名的名字同名的文件夹，此时可以运行了。

```
caolr@caolr-u:~/ep-hpl/hpl-2.3$ cd bin/ep
caolr@caolr-u:~/ep-hpl/hpl-2.3/bin/ep$ mpirun -np 4 ./xhpl > HPL-Benchmark.txt
```

没有报错即运行成功。运行结果保存在当前这个路径下，与代码中同名的 txt 中，查看即可。



⑧注意：在复制 Make.Linux_PII_FBLAS 到上层目录重命名时，对于命名规则和 cpu 架构有关，可以通过命令 cat /proc/cpuinfo 来查看，否则会发生如下错误：

```
/usr/bin/ld: cannot open output file /usr/local/HPL/hpl-2.3/bin/jyq/xhpl: Permis
sion denied
collect2: error: ld returned 1 exit status
Makefile:76: recipe for target 'dexe.grd' failed
make[2]: *** [dexe.grd] Error 1
make[2]: Leaving directory '/usr/local/HPL/hpl-2.3/testing/ptest/jyq'
Make.top:64: recipe for target 'build_tst' failed
make[1]: *** [build_tst] Error 2
make[1]: Leaving directory '/usr/local/HPL/hpl-2.3'
Makefile:72: recipe for target 'build' failed
make: *** [build] Error 2
```

3. 运行并进行测试

使用 lshw、lscpu 等命令查看各自电脑配置，结果如下。

Item	**Configuration**
Computer1（金玉卿）	CPU: Intel Core i5-2410M*4, 2.30GHz
	Memory: 9.6G
	Hard disk: 491.2G
	Cacheline（L2）：256K
Computer2（郭真铃）	CPU:Intel Core i7-8550U CPU*8 @ 1.80GHz
	Memory: 7826MB（约 8G）
	Hard disk: 80G
	Cacheline（L2）：256K
Computer3（曹璐然）	CPU: Pentium® Dual-Core CPU E5800 @3,2GHz *2
	Memory: 1992M
	Hard Disk: 465G
	Cacheline(L2): 2048K

Item	**Description**	**Version**

OS	Ubuntu	18.04
Compiler	GCC	7.5.0
MPI	MPI	MPICH-3.3.2
BLAS	BLAS	BLAS-3.8.0
HPL	HPL	HPL-2.3

运行时进入 bin 目录中生成的项目文件夹，执行 mpirun -np x ./xhpl > [文件名]命令，此处 x 表示线程数，若小于某一组 PQ 的值会报错；文件名可自定义方便之后的浏览。生成文件保存在项目文件夹中。

4. 优化（详细阐述如何进行优化）并进行测试

4.1 HPL.dat 中参数的优化

HPL.dat 文件内容如下：

```
HPLinpack benchmark input file                                  #1
Innovative Computing Laboratory, University of Tennessee        #2
HPL.out      output file name (if any)                          #3
6            device out (6=stdout,7=stderr,file)                #4
1            # of problems sizes (N)                            #5
5000         Ns                                                 #6
1            # of NBs                                           #7
128          NBs                                                #8
0            PMAP process mapping (0=Row-,1=Column-major)       #9
3            # of process grids (P x Q)                         #10
2 1 4        Ps                                                 #11
2 4 1        Qs                                                 #12
16.0         threshold                                          #13
3            # of panel fact                                    #14
0 1 2        PFACTs (0=left, 1=Crout, 2=Right)                  #15
2            # of recursive stopping criterium                  #16
2 4          NBMINs (>= 1)                                      #17
1            # of panels in recursion                           #18
2            NDIVs                                              #19
3            # of recursive panel fact.                         #20
0 1 2        RFACTs (0=left, 1=Crout, 2=Right)                  #21
1            # of broadcast                                     #22
0            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)       #23
1            # of lookahead depth                               #24
0            DEPTHs (>=0)                                       #25
2            SWAP (0=bin-exch,1=long,2=mix)                     #26
64           swapping threshold                                 #27
0            L1 in (0=transposed,1=no-transposed) form          #28
0            U    in (0=transposed,1=no-transposed) form        #29
```

1	Equilibration (0=no,1=yes)	#30
8	memory alignment in double (> 0)	#31

下面逐个简要说明每个参数的含义及一般配置：

（1）第 1、2 行为注释说明行，不需要作修改

（2）第 3、4 行说明输出结果文件的形式

“device out”为“6”时，测试结果输出至标准输出(stdout)

“device out”为“7”时，测试结果输出至标准错误输出(stderr)

“device out”为其他值时，测试结果输出至第三行所指定的文件中

（3）第 5，6 行说明求解矩阵的次数和大小 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。因此，对于一般系统而言，要尽量增大矩阵规模 N 的同时，又要保证不使用系统缓存。因为操作系统本身需要占用一定的内存，除了矩阵(N×N)之外，HPL 还有其他的内存开销，另外通信也需要占用一些缓存。矩阵占用系统总内存的 80%左右为最佳，即 N×N×8=系统总内存×80%。

（4）第 7、8 行说明求解矩阵分块的大小 NB

为提高数据的局部性，从而提高整体性能，HPL 采用分块矩阵的算法。分块的大小对性能有很大的影响，NB 的选择和软硬件许多因素密切相关。

下面给出一些平台上的 NB 选择。

平台	L2 Cache	运行模式	数学库	NB
Intel P4 Xeon	512KB	32 位	ATLAS	400
			MKL	384
			GOTO	192
Intel P4 Xeon (Nocona)	1MB	64 位	GOTO	96
AMD Opteron	1MB	64 位	GOTO	232

NB 值的选择主要是通过实际测试得到最优值。但 NB 的选择上还是有一些规律可寻，如:NB 不可能太大或太小，一般在 256 以下:NB×8 一定是 Cache line 的倍数等。

另外，NB 大小的选择还跟通信方式、矩阵规模、网络、处理器速度等有关系。一般通过单节点或单 CPU 测试可以得到几个较好的 NB 值，但当系统规模增加、问题规模变大，有些 NB 取值所得性能会下降。所以最好在小规模测试时选择 3 个左右性能不错的 NB，再通过大规模测试检验这些选择。

（5）第 9 行是选择处理器阵列是按列的排列方式还是按行的排列方式。

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。

（6）第 10-12 行说明二位处理器网络（P×Q），二位处理器网格（P×Q）的有以下几

个要求：

P×Q=进程数，这是 HPL 的硬性规定；

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。

当 Q/4≤P≤Q 时，性能最优。

P≤Q：一般来说，P 的值尽量取得小一点，因为列项通信量要远大于横向通信。

$P=2^n$，即 P 最好选择 2 的幂次，HPL 中，L 分解的列项通信采用二元交换法，当列项处理器个数 P 为 2 的幂时，性能最优。

（7）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。

（8）第 14-21 行指明 L 分解的方式。

在 HPL 官方文档中，推荐的设置为：

```
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)
```

（9）第 22、23 行说明 L 的横向广播方式。

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。

推荐的配置为：

```
2            # of broadcast
1 3          BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
```

（10）第 24、25 行说明横向通信的通信深度，依赖于机器的配置和问题规模的大小。

推荐配置为：

```
2            # of lookahead depth
0 1          DEPTHs (>=0)
```

（11）第 26、27 行说明 U 的广播算法。

推荐配置为：

```
2            SWAP (0=bin-exch,1=long,2=mix)
60           swapping threshold
```

（12）第 28、29 行分别说明 L 和 U 的数据存放格式。

推荐配置为：

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

（13）第 30 行主要在回代中使用，一般使用其默认值。

（14）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其地址，处于安全考虑，可以选择 8。

4.2 其他性能优化

①MPI

对于常用的 MPICH 来说，安装编译 MPICH 时，使其节点内采用共享内存进行通信可以提升一部分性能，在 configure 时，设置”—with-comm=shared”。

对于 GM 来说，在找到路由以后，将每个节点的 gm_mapper 进程 kill 掉，大概有一个百分点的性能提高。

②操作系统

操作系统层的性能优化方法，如裁剪内核、改变页面大小、调整内核参数、调整网络参数等等。

实验数据：

优化前三台设备数据如下：

	进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）
computer1	1	35	4	1	1	0.00	6.8213e-01
	2	35	4	1	2	0.00	3.3228e-01
	4	35	4	1	4	0.00	3.0969e-01
	8	35	4	1	8	0.10	3.0428e-04
	16	35	4	1	16	0.17	1.6630e-04
computer2	进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）
	1	35	4	1	1	0.00	9.4514e-01
	2	35	4	1	2	0.00	5.4761e-01
	4	35	4	1	4	0.00	8.5634e-01
	8	35	4	1	8	0.00	2.6093e-01
	16	35	4	1	16	0.13	2.3769e-04
computer3	进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）
	1	35	4	1	1	0	9.8149E-01
	2	35	4	1	2	0	4.0378E-01
	4	35	3	1	4	0.21	5.9397E-04
	8	35	4	1	8	0.12	2.5354E-04
	16	35	4	1	16	0.51	8.8997E-05

优化后最优数据如下：

	进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）	性能提升（倍）
computer1	2	2000	32	1	2	0.89	6.5645e+00	9.62
computer2	4	5000	16	2	2	7.17	1.1631e+01	12.3
computer3	2	840	64	1	2	0.10	3.7885	9.38

实验数据处理：

Computer1（金玉卿）：

3. 不进行优化，分析在什么情况下（N、NB、P、Q 等），可以获得更好的性能。

在未进行优化的前提下，只修改进程数和 PQ 的值。进程数分别取 1、2、4、8、16，二位处理器网格（P×Q）的要求如下：

P×Q=进程数；

P≤Q；一般来说，P 的值尽量取得小一点，因为列项通信量要远大于横向通信。

$P=2^n$，即 P 最好选择 2 的幂次，HPL 中，L 分解的列项通信采用二元交换法，当列项处理器个数 P 为 2 的幂时，性能最优。因此可以设计进程数对应的 P 和 Q 的取值如下表：

进程数	P	Q
1	1	1
2	1	2
4	1	4
	2	2
8	1	8
	2	4
16	1	16
	2	8
	4	4

N 为默认的 29、30、34、35，NB 为默认的 1、2、3、4，在不同线程时，得到的最佳 Gflops 如下：

进程数	N	NB	P	Q	执行时间	HPL 测试所得的实际

					（s）	峰值速度（Gflops）
1	35	4	1	1	0.00	6.8213e-01
2	35	4	1	2	0.00	3.3228e-01
4	35	4	1	4	0.00	3.0969e-01
8	35	4	1	8	0.10	3.0428e-04
16	35	4	1	16	0.17	1.6630e-04

从测试结果表中可以看出，在 N 的值为 29、30、34、35 的情况下，一直是 N=35 获得最优值，从目前结果推测 N 似乎越大越好。电脑配置 cpu 核数为 4，在进程数超过电脑核数时，运行效果明显变差，在 16 进程下花费的时间较久，且测得的实际峰值速度最低，综上所述，在 N=35,NB=4,P=1,Q=1 时取得最好的性能，实际峰值速度达到 0.58213Gflops

4. 展示优化后的测试结果，并详细分析调整相关参数或优化程序代码后能够获得更高的性能的原因。

从未优化时的测试数据可以推测 N 和 NB 的值似乎越大越好，同时进程数尽量不超过 cpu 的核数，即一个进程对应一个 CPU 获得的性能较好。

对于 NB 的取值，主要是通过实际测试得到最优值，但由经验可知，NB 不能太大或太小，一般在 256 以下；NB×8=cacheline 的倍数，通过单 CPU 测试可以得到较好的 NB 值，所以在这里取 N=200，P=1，Q=1，由于 cacheline=256K，所以预取 NB 的值为 32，64，96，128，160，192，进行测试，测试结果如下表：

进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）
1	200	32	1	1	0.00	2.8736e+00
1	200	64	1	1	0.00	2.6701e+00
1	200	96	1	1	0.00	2.5779e+00
1	200	128	1	1	0.00	2.4856e+00
1	200	192	1	1	0.00	2.4256e+00
1	200	256	1	1	0.00	2.4812e+00

由上表可知，NB 取 32，64，96 时，性能较好。

对于 N 的取值，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。因此，对于一般系统而言，要尽量增大矩阵规模 N 的同时，又要保证不使用系统缓存。因为操作系统本身需要占用一定的内存，除矩阵（N×N）外，HPL 有其他的内存开销，同时通信也会占用一些缓存。矩阵占用系

统总内存的 80%左右为最佳，即 N×N×8=系统内存（byte）×80%。

由内存大小 9.6G 大概计算出 N 的大小为 30983，此时固定 P=1、Q=4 或 P=1、Q=2，NB=64，调整 N 的大小，对 N 取值为 2000，3000，4000，5000，6000，进行测试，测试结果如下表：

进程数	N	NB	P	Q	执行时间（s）	Gflops
4	2000	64	1	4	1.39	3.8346e+00
4	3000	64	1	4	6.45	2.7909e+00
4	4000	64	1	4	17.29	2.4686e+00
4	5000	64	1	4	35.52	2.3469e+00
4	6000	64	1	4	61.89	2.3276e+00
4	1500	64	1	4	1.54	1.4646e+00
4	1200	64	1	4	0.93	1.2443e+00
4	1800	64	1	4	1.04	3.7286e+00
4	2200	64	1	4	2.22	3.2013e+00
4	2300	64	1	4	2.57	3.1584e+00

从上述表格可知，在 N 为 2000 左右时实际峰值速度较大，综合测试表格如下：

进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）
2	1900	32	1	2	0.77	6.1719e+00
2	2000	32	1	2	0.89	6.5645e+00
2	2100	32	1	2	0.89	6.1799e+00
2	1900	64	1	2	0.79	5.9815e+00
2	2000	64	1	2	0.93	5.8183e+00
2	2100	64	1	2	1.09	5.7958e+00
4	1950	32	1	4	0.84	5.9880e+00
4	2000	32	1	4	0.91	5.9802e+00
4	2050	32	1	4	1.00	5.8315e+00
4	2100	32	1	4	1.05	5.9727e+00

由上表可知，当 N=2000，NB=32，P=1，Q=2 时，实际峰值速度最大为 6.5645Gflops。与优化前的最好结果 0.68213Gflops 比较，性能提高了 n=9.62 倍。

Computer2（郭真铃）：

1. 不进行优化，分析在什么情况下（N、NB、P、Q 等），可以获得更好的性能。

进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）
1	35	4	1	1	0.00	9.4514e-01
2	35	4	1	2	0.00	5.4761e-01
4	35	4	1	4	0.00	8.5634e-01
8	35	4	1	8	0.00	2.6093e-01
16	35	4	1	16	0.13	2.3769e-04

以上的结果是从各种进程数中挑选 Gflops 最大的结果。各个测试结果除去 P、Q 不同外，N、NB 的值的设置是一样的，都是 N：29、30、34、35；NB：1、2、3、4；而 P、Q 对应进程数，分别设置如下（表 2）：

进程数	P	Q
1	1	1
2	1	2
4	1	4
	2	2
8	1	8
	2	4
16	1	16
	2	8
	4	4

从测试结果表中可以看出，在 N 分别为 29、30、34、35 的情况下，一直是 N=35 获得最优值，从目前结果可见 N 似乎越大越好。在 NB 分别为 1、2、3、4 的情况下，一直是 NB=4 获得最优值，从目前结果可见 NB 似乎越大越好。结合测试结果表和表 2，一直是 P 为 1 的情况最优。总而言之，在未优化时，设定 N=35、NB=4、P=1、Q=1，有结果 Gflops =9.4514e-01。

2. 展示优化后的测试结果，并详细分析调整相关参数或优化程序代码后能够获得更高的性能的原因。

从上面的结果中可知进程数为 1、4 的结果相对较优，因此选取进程数分别为 1、4 两种情况来进行测试。

其次确定 N 值，根据他人经验可知，有 N×N×8=系统内存（以 Byte 为单位）×80%，根据我的硬件情况得到 N 要小于等于 28646，此时固定 P=1、Q=1、NB=4，调整 N，最终发现 N=5000 有较优值。

然后设定 N=5000，使 NB 分别为 4、8、16、32、64、128；P*Q 分别为 1、4，结果如下：

进程数	N	NB	P	Q	执行时间（s）	Gflops
1	5000	4	1	1	30.48	2.7349e+00
1	5000	8	1	1	30.47	2.7363e+00
1	5000	16	1	1	30.46	2.7370e+00
1	5000	32	1	1	28.82	2.8931e+00
1	5000	64	1	1	26.07	3.1976e+00
1	5000	128	1	1	23.11	3.6081e+00
4	5000	4	1	4	11.82	7.0536e+00
4	5000	8	1	4	7.99	1.0433e+01
4	5000	16	1	4	7.72	1.0793e+01
4	5000	32	1	4	7.66	1.0890e+01
4	5000	64	1	4	8.64	9.6540e+00
4	5000	128	1	4	12.32	6.7698e+00
4	5000	4	2	2	11.94	6.9823e+00
4	5000	8	2	2	7.39	1.1285e+01
4	5000	16	2	2	7.17	1.1631e+01
4	5000	32	2	2	7.35	1.1344e+01
4	5000	64	2	2	7.44	1.1204e+01
4	5000	128	2	2	8.59	9.6999e+00

从上述表格可知，设定 N=5000、NB=16、P=2、Q=2 时，可得到最优结果 Gflops=1.1631e+01，相较于未优化前 Gflops =9.4514e-01，性能提高了 n=12.3 倍。

Computer3（曹璐然）：

1.不进行优化，分析在什么情况下（N、NB、P、Q 等），可以获得更好的性能。

首先在未进行优化的前提下，只是修改进程数和 PQ 的值。进程数分别取 1、2、4、8、16 时，P、Q 根据分析中的规则取值为：

进程数	P	Q
1	1	1
2	1	2
4	1	4
	2	2
8	1	8
	2	4
16	1	16
	2	8
	4	4

N 为默认的 29、30、34、35，NB 为默认的 1、2、3、4。不同线程时，得到的最佳 Gflops 如下：

进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）
1	35	4	1	1	0	9.8149E-01
2	35	4	1	2	0	4.0378E-01
4	35	3	1	4	0.21	5.9397E-04
8	35	4	1	8	0.12	2.5354E-04
16	35	4	1	16	0.51	8.8997E-05

通过上表再结合我的 cpu 是两核可知，当进程数超过 cpu 核数的时候效果明显变差。甚至我的 16 线程根本跑不完。所以在线程为 1、2 时表现较好。此时的 P、Q 相乘应为线程数，且 P 为较小值，可以得到较好的 Gflops 值。

2.展示优化后的测试结果，并详细分析调整相关参数或优化程序代码后能够获得更高的性能的原因。

结合我的设备 cpu 核数为 2，所以 P、Q 取 1、1 和 1、2。根据未优化所得结果可以看出有 N、NB 更大时会有更好的结果的趋势。所以接下来寻找 N、NB 的最大值。

首先固定其他数据，测试一些 NB 的值，找出比较合适 NB。矩阵分块的大小对计算的影响较大，分块过大容易造成过载不平衡；分块过小通讯开销会很大同样影响性能。我这里根据所谓的 NB*8=Cacheline 的倍数这个经验，测试了一些数据结果如下：

进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）

2	50	64	1	2	0	1.0676
2	50	96	1	2	0	1.0806
2	50	128	1	2	0	1.0556
2	50	160	1	2	0	1.0496
2	50	192	1	2	0	1.0496
2	50	256	1	2	0	1.0466

可以看出 NB 取 64、96、128 时，表现较好，保留这三个值带入到后面的测试。

最后寻找 N，因为 N 较大所以放在后面比较节省测试的时间。N 越大时，有效计算所占的比例也越大，系统浮点处理性能也就越高；但是过大之后内存不够用，会发生数据交换，磁盘交换区的性能远低于内存，反而降低了有效计算的比例。经验公式指出 N 的理论巅峰值为 N*N*8=内存*80%，根据此公式可以算出 N≈15764。但是发现在实际实验中取不到这么大，下面是对 N 进行一些取值发现的比较接近最大值的情况：

进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）
1	2000	128	1	1	4.25	1.2566
1	3000	128	1	1	19.52	9.2322E-01
2	2000	128	1	2	3.24	1.6464
2	3000	128	1	2	13.60	1.3244
2	1500	128	1	2	1.18	1.9019
2	500	128	1	2	0.03	2.9724
2	700	128	1	2	0.08	3.0440
2	800	128	1	2	0.11	3.2292
2	850	128	1	2	0.14	3.0397
2	900	128	1	2	0.15	3.2070
2	950	128	1	2	0.22	2.6619
2	1000	128	1	2	0.23	2.9597

由上表可以看出 N 在 800 左右且线程等于 cpu 核数时 Gflops 比较高，下面综合各项数据再进行一次测试，其中得到的一些较好结果如下表：

进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）
2	750	64	1	2	0	3.6489

2	750	96	1	2	0.08	3.5078
2	760	64	1	2	0.08	3.6770
2	780	96	1	2	0.09	3.4295
2	800	64	1	2	0.10	3.7663
2	820	64	1	2	0.12	3.7282
2	840	32	1	2	0.11	3.6405
2	840	64	1	2	0.10	3.7885
2	870	64	1	2	0.12	3.7308
2	900	64	1	2	0.13	3.7686

由上表可知，当 N=840，NB=64，P=1，Q=2 时，Gflops 达到巅峰为 3.7885。与优化前的最好结果 4.0378E-01 比较，性能提高了 n=9.38 倍。

实验结果与分析：

优化结果如下，分析已在步骤和数据处理中进行：

	进程数	N	NB	P	Q	执行时间（s）	HPL 测试所得的实际峰值速度（Gflops）	优化倍数
computer1								
改进前	1	35	4	1	1	0.00	6.8213e-01	9.62
改进后	2	2000	32	1	2	0.89	6.5645e+00	
computer2								
改进前	1	35	4	1	1	0.00	9.4514e-01	12.3
改进后	4	5000	16	2	2	7.17	1.1631e+01	
computer3								
改进前	1	35	4	1	1	0	9.8149E-01	9.38
改进后	2	840	64	1	2	0.10	3.7885	

教师评审

教师评语	实验成绩
签名： 日期：	