



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 184

| 组号 | 学号 | 姓名 | 贡献比(%) |
|----|----------|-----|--------|
| | 41824179 | 王丹琳 | |
| | | | |
| | | | |
| | | | |
| | | | |

指 导 老 师：

实 验 学 时：4 学时

时 间：2021 年 4 月 25 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) 下载 MPI 安装包并解压

在官网 <http://www.mpich.org/downloads/> 下载 mpich 的最新版本(本次使用的是 3.3.2)。

在下载安装包目录下可以找到已经下载的 mpich-3.3.2.tar.gz，在该目录下打开终端使用下面命令解压压缩包：

```
tar -zxvf mpich-3.3.2.tar.gz
```

进入压缩目录下：

```
cd mpich-3.3.2
```

(2) 配置编译环境并进行安装

```
./configure --prefix=/program_file/ mpich-3.3.2
```

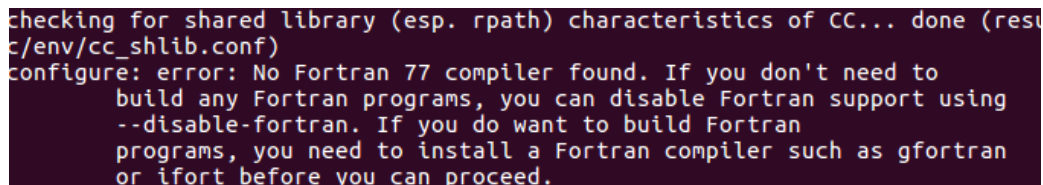
其中 /program_file/ mpich-3.3.2 为安装的路径，可以自由选择路径，该路径需要在下面的 HPL 配置中用到，这里需要将其记忆。

接下来进行安装，命令如下：

```
sudo make
```

```
sudo make install
```

Ubuntu 环境会遇到如下问题：



```
checking for shared library (esp. rpath) characteristics of CC... done (result: c/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.
```

图 1 执行安装命令所遇到的问题

这里可能是因为缺少安装 C、C++、F77 和 F90 编译器，所以如果遇到以上问题，可以先进行如下命令。

对于 Ubuntu 可以使用下面命令进行补充安装

```
sudo apt-get install fort77
```

```
sudo apt-get install gfortran
```

对于 Centos 可以使用下面命令进行补充安装。

```
yum install gcc -y
```

```
yum install gcc-c++ -y
```

```
yum install gcc-gfortran -y
```

再执行上面的安装命令，完成 MPI 的安装

```
sudo make
```

```
sudo make install
```

（3）环境配置

使用 vi 或者 vim 打开 bashrc 文件，添加内容

```
vim ~/.bashrc
```

在最后一行添加如下内容（按 i 可以对 vim 进行编辑，编辑完后，按 esc 键再输入:wq 退出，vim 的学习链接 <http://www.runoob.com/linux/linux-vim.html>）

```
export PATH=/program_file/mpich-3.3.2/bin:$PATH
```

```
export LD_LIBRARY_PATH=/program_file/mpich-3.3.2/lib:$LD_LIBRARY_PATH
```

编辑完成后输入下面内容激活环境变量

```
source ~/.bashrc
```

（4）测试 MPI 是否安装成功

检测命令导出是否成功

```
which mpirun
```

如果显示出安装路径（如下）则说明安装成功

```
loongson@loongson-VirtualBox:~/下载/mpich-3.3.2$ which mpirun
/usr/bin/mpirun
loongson@loongson-VirtualBox:~/下载/mpich-3.3.2$
```

图 2 MPI 安装成功的验证

在下载压缩包中有提供测试的样例代码，进入到压缩包路径下执行下面命令

```
cd examples
```

```
mpicc hellow.c -o hellow
```

```
mpirun -np 4 ./hellow
```

如果运行结果如下证明安装已经完成。

```
loongson@loongson-VirtualBox:~/下载/mpich-3.3.2/examples$ mpirun -np 4 ./hellow
Hello world from process 2 of 4
Hello world from process 1 of 4
Hello world from process 0 of 4
Hello world from process 3 of 4
```

图 3 4 线程 hellow 程序运行结果

2. 配置 HPL

（1）安装 GotoBLAS2

首先在官网下载（本次使用的是 GotoBLAS2-1.1.3），官网地址：

<https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2>

首先对压缩文件进行解压并进入该目录：

```
tar -xzvf GotoBLAS2-1.13.tar.gz
```

```
cd GotoBLAS2-1.13
```

进行快速安装：

```
./quickbuild.64bit
```

如果出现大量 kernel/x86_64/gemm_ncopy_4.S 的错误（如下图）

```

../kernel/x86_64/gemm_ncopy_4.S: Assembler messages:
../kernel/x86_64/gemm_ncopy_4.S:175: 错误:  invalid operands (*UND* and *ABS* se
ctions) for `*'
../kernel/x86_64/gemm_ncopy_4.S:176: 错误:  invalid operands (*UND* and *ABS* se
ctions) for `*'
../kernel/x86_64/gemm_ncopy_4.S:177: 错误:  invalid operands (*UND* and *ABS* se
ctions) for `*'
../kernel/x86_64/gemm_ncopy_4.S:178: 错误:  invalid operands (*UND* and *ABS* se
ctions) for `*'
../kernel/x86_64/gemm_ncopy_4.S:180: 错误:  invalid operands (*UND* and *ABS* se
ctions) for `*'
../kernel/x86_64/gemm_ncopy_4.S:328: 错误:  invalid operands (*UND* and *ABS* se
ctions) for `*'
../kernel/x86_64/gemm_ncopy_4.S:329: 错误:  invalid operands (*UND* and *ABS* se
ctions) for `*'
../kernel/x86_64/gemm_ncopy_4.S:331: 错误:  invalid operands (*UND* and *ABS* se
ctions) for `*'
Makefile.L3:353: recipe for target 'sgemm_ncpy.o' failed
make[1]: *** [sgemm_ncpy.o] Error 1
make[1]: *** 正在等待未完成任务....
make[1]: Leaving directory '/home/loongson/下载/GotoBLAS2-1.13/kernel'
Makefile:89: recipe for target 'libs' failed
make: *** [libs] Error 1

```

图 4 GotoBLAS2 安装错误的截图

则需要添加编译选项（这个需要根据个人计算机 NEHALEM 是 Intel 的）：

```
make clean
```

```
make CC=gcc BINARY=64 TARGET=NEHALEM
```

如果再遇到类似如下报错

```

/usr/lib64/gcc/x86_64-suse-linux/4.7/../../../../x86_64-suse-linux/bin/ld: cannot find -l-l
collect2: error: ld returned 1 exit status

make[1]: * [../libgoto2_nehalemp-r1.13.so] Error 1

make[1]: Leaving directory `/home/ken/bin/build/GotoBLAS2/exports' make:

[shared] Error 2

```

```

-lc && echo OK.
/usr/bin/ld: 找不到 -l-l
collect2: error: ld returned 1 exit status
Makefile:100: recipe for target '../libgoto2_nehalemp-r1.13.so' failed
make[1]: *** [../libgoto2_nehalemp-r1.13.so] Error 1
make[1]: Leaving directory '/home/loongson/下载/GotoBLAS2-1.13/exports'
Makefile:49: recipe for target 'shared' failed
make: *** [shared] Error 2

```

图 5 GotoBLAS2 安装错误的截图

则需要修改 f_check: 进入 f_check

```
vim f_check
```

修改第 298 行（如下图）：

```
print MAKEFILE "FEXTRALIB=$linker_L -lgfortran -lm -lquadmath -lm $linker_a\n";
```

再进行编译

```
make clean
```

```
make CC=gcc BINARY=64 TARGET=NEHALEM
```

(2) 配置 HPL

首先要下载 hpl 的安装包 (本次使用 hpl-2.3), 官方网址:

<http://www.netlib.org/benchmark/hpl/hpl-2.3.tar.gz>

将下载内容进行解压

```
tar -xzvf hpl-2.3.tar.gz
```

```
cd hpl-2.3
```

编辑 Make 配置文件

```
vim Make.name_str
```

需要编写的内容如下, 标红色是有可能需要改动的内容

TOPdir 是 hpl 解压缩后所在的路径

MPdir 是 MPI 的安装路径

LAdir 是 GotoBLAS2 解压缩后所在路径

```
SHELL      = /bin/sh
CD          = cd
CP          = cp
LN_S        = ln -s
MKDIR       = mkdir
RM          = /bin/rm -f
TOUCH       = touch
ARCH        = name_str
TOPdir      = /program_file/ hpl-2.3
INCdir      = $(TOPdir)/include
BINdir      = $(TOPdir)/bin/$(ARCH)
LIBdir      = $(TOPdir)/lib/$(ARCH)
HPLlib      = $(LIBdir)/libhpl.a
MPdir       = /program_file/ mpich-3.3.2
```

```
MPinc      = -I$(MPdir)/include
MPlib      = $(MPdir)/lib/libmpich.so
LAdir      = /program_file/ GotoBLAS2
LAinc      =
LAlib      = $(LAdir)/libgoto2.a
F2CDEFS    =
HPL_INCLUDES = -I$(INCdir) -I$(INCdir)/$(ARCH) $(LAinc) $(MPinc) -lpthread
HPL_LIBS    = $(HPLlib) $(LAlib) $(MPlib) -lpthread
HPL_OPTS    = -DHPL_CALL_CBLAS
HPL_DEFS    = $(F2CDEFS) $(HPL_OPTS) $(HPL_INCLUDES)
CC          = $(MPdir)/bin/mpicc -lpthread
CCNOOPT     = $(HPL_DEFS)
CCFLAGS     = $(HPL_DEFS) -fomit-frame-pointer -O3 -funroll-loops
LINKER      = $(MPdir)/bin/mpif77
LINKFLAGS   = $(CCFLAGS)
ARCHIVER    = ar
ARFLAGS     = r
RANLIB      = echo
```

编写好后保存退出，执行：

```
make arch=name_str
```

如果配置文件没有问题并且，其他的两个已经安装成功执行完之后会出现 bin 目录，bin 目录中有 name_str 文件夹：

```
cd bin
cd name_str
ls
```

此时应该会看到两个文件 HPL.dat, xhpl



图 6 make 结果

进行初步尝试

```
mpirun -np 4 ./xhpl
```

如果运行正确会显示出类似下面的结果，

```
=====
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)= 1.88391027e-02 ..... PASSED
=====
T/V          N    NB    P    Q          Time          Gflops
-----
WR00R2R4      35     4     4     1          0.88          3.4575e-05
HPL_pdgesv() start time Sun Apr 25 09:36:33 2021
HPL_pdgesv() end time   Sun Apr 25 09:36:34 2021
=====
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)= 2.80968060e-02 ..... PASSED
=====
Finished      864 tests with the following results:
              864 tests completed and passed residual checks,
               0 tests completed and failed residual checks,
               0 tests skipped because of illegal input values.
=====
End of Tests.
```

图 7 HPL 初次测试结果

3. 运行并行测试

1) 单机测试

修改 $P \times Q$ 的大小，然后依次输入 `mpirun -np 4 ./xhpl`，`mpirun -np 8 ./xhpl`，`mpirun -np 16 ./xhpl` 进行测试，并记录结果

2) 集群测试

将组内成员的电脑构建为小“集群”，测试该“集群”的性能

本组实现的 MPI 集群搭建是两台计算机在同一个局域网下搭建的。如果没有公共的在同一个局域网下，可以通过内网穿透然后再进行连接，在搭建集群之前，首先要保证，两台计算机已经安装了 MPICH 并且版本最好一致。此外除了 MPICH 版本一致外，GCC 以及 fortran 的版本最好也要相同，防止不能运行共同的文件，最好的方法就是使用两个一样 Linux 版本的计算机测试。

在测试之前，需要保证两个计算机的用户名是相同的，不相同最好改一下用户名，因为后面每个虚拟机节点之间免密登录时需要保证用户名一致，这样可以避免出现未知的问题。

具体的安装方法：

1) 两台计算机安装 SSH，并设置 SSH 免密登录

使用 `ifconfig` 命令查看两个计算机的 IP 地址，修改两个计算机的 hosts 文件，在两个计算机的 hosts 文件分别添加节点 IP。注意：由于每个机器的 ip 会自动分配发生变化，所以在在此之前，我们已经完成了虚拟机的静态 ip 设置


```
sudo vim /etc/hosts
```

```
192.168.43.5 node1
```

```
192.168.43.7 node2
```

我们可以通过 ping 命令判断是否机器可以 ping 到，来检查是否处于同一个局域网内，若出现错误了，需要检查 hosts 文件是否修改正确。

```
ping node1
```

```
ping node2
```

```
yyc@yyc-VirtualBox:~$ ping node2
PING node2 (192.168.43.7) 56(84) bytes of data.
64 bytes from node2 (192.168.43.7): icmp_seq=1 ttl=64 time=6.52 ms
64 bytes from node2 (192.168.43.7): icmp_seq=2 ttl=64 time=26.2 ms
64 bytes from node2 (192.168.43.7): icmp_seq=3 ttl=64 time=48.6 ms
64 bytes from node2 (192.168.43.7): icmp_seq=4 ttl=64 time=276 ms
64 bytes from node2 (192.168.43.7): icmp_seq=5 ttl=64 time=86.5 ms
64 bytes from node2 (192.168.43.7): icmp_seq=6 ttl=64 time=225 ms
64 bytes from node2 (192.168.43.7): icmp_seq=7 ttl=64 time=143 ms
64 bytes from node2 (192.168.43.7): icmp_seq=8 ttl=64 time=163 ms
64 bytes from node2 (192.168.43.7): icmp_seq=9 ttl=64 time=188 ms
64 bytes from node2 (192.168.43.7): icmp_seq=10 ttl=64 time=85.1 ms
64 bytes from node2 (192.168.43.7): icmp_seq=11 ttl=64 time=28.0 ms
64 bytes from node2 (192.168.43.7): icmp_seq=12 ttl=64 time=50.8 ms
```

图 8 node1 ping node2 结果

在两个节点分别安装 ssh:

```
sudo apt-get install ssh
```

此时，可以先尝试是否可以通过 ssh 命令进行各计算机的远程登录，若已经实行了免密登录则跳过下面几步，但是一般情况下，此时的登录是需要密码的。

```
ssh yyc@node1
```

```
ssh yyc@node2
```

```
yyc@yyc-VirtualBox:~$ ssh yyc@node2
The authenticity of host 'node2 (192.168.43.7)' can't be established.
ECDSA key fingerprint is SHA256:GvonwEyTZT5od0OvdeXIWzJKhPQ95Rr+2ahL7doj0Qc.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'node2,192.168.43.7' (ECDSA) to the list of known
ts.
yyc@node2's password:
Welcome to Ubuntu 16.04.7 LTS (GNU/Linux 4.15.0-142-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

28 个可升级软件包。
0 个安全更新。

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.
```

图 9 node1 通过 ssh 命令进行 node2 远程登录结果

生成 ssh 的公钥和私钥，并将两个节点的公钥都添加到自己的 `authorized_keys` 中，以此实现免密登录了。

具体操作如下：

node1 ， node 2 分别输入以下命令，生成公钥和私钥（在.ssh 文件中）

```
ssh-keygen -t rsa
```

```

yyc@yyc-VirtualBox:~$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/yyc/.ssh/id_rsa):
Created directory '/home/yyc/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/yyc/.ssh/id_rsa.
Your public key has been saved in /home/yyc/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:6d3KCWEuEGpL88z8P/V8aled0pa3iS2B6xQjVLj8Y50 yyc@yyc-VirtualBox
The key's randomart image is:
+---[RSA 2048]---+
  .                .
  .                .
  .   .   .   .   .
  .   .+ E .
  = .   S.oo+   o
  o B . + oo+o. .o
  . = . +..+o +.
  .   ..O.++*=..
  .   ....=oo=++
+---[SHA256]---+

```

图 9 node1 生成公钥和私钥

在 node1 与 node2 输入如下命令，完成对自己计算机的认证，该操作完成后，使用 ssh 可以对自己免密登录

```
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

node1 节点进行如下操作，将 node1 的公钥传到 node2

```
scp ~/.ssh/id_rsa.pub yyc@node2:~/.ssh/id_rsa.pub.node1
```

```

Enter passphrase for key '/home/yyc/.ssh/id_rsa':
id_rsa.pub          100% 400    0.4KB/s   00:00

```

图 10 node1 的公钥传到 node2

node2 节点进行如下操作，将刚刚传到的 node1 公钥添加认证

```
cat ~/.ssh/id_rsa.pub.node1 >> ~/.ssh/authorized_keys
```

node2 节点进行如下操作，将 node1 的公钥传到 node1

```
scp ~/.ssh/id_rsa.pub 用户名@node1:~/.ssh/id_rsa.pub.node2
```

node1 节点进行如下操作，将刚刚传到的 node2 公钥添加认证

```
cat ~/.ssh/id_rsa.pub.node2 >> ~/.ssh/authorized_keys
```

再次使用 ssh 命令测试是否 node1 和 node2 之间都可以进行免密登录。

```
ssh yyc@node1
```

```
ssh yyc@node2
```

如果可以相互连接成功,则表示 node1 和 node2 之间通过 ssh 可以进行免密登录,那么就可以尝试集群共同运行程序进行尝试。

在测试之前,首先需要在两个计算机上创建一个共享的目录:在 node1,node2 分别创建 /home/mpi_share 文件夹,然后加入 cpi 程序,再同加入 mpi_config 文件,作为运行的配置文件,内容如下:

```
node1:4
```

```
node2:4
```

这里的添加根据自己的 CPU 核心数进行添加,然后在任意节点进行运行:

```
mpirun -n 80 -f ./mpi_config ./cpi
```

运行结果如下:

```
root@yyc-VirtualBox:/mpi_share# mpirun -n 80 -f ./mpi_config ./cpi
Process 53 of 80 is on yyc-VirtualBox
Process 45 of 80 is on yyc-VirtualBox
Process 4 of 80 is on yyc-VirtualBox
Process 6 of 80 is on yyc-VirtualBox
Process 77 of 80 is on yyc-VirtualBox
Process 13 of 80 is on yyc-VirtualBox
Process 36 of 80 is on yyc-VirtualBox
Process 55 of 80 is on yyc-VirtualBox
Process 5 of 80 is on yyc-VirtualBox
Process 29 of 80 is on yyc-VirtualBox
Process 30 of 80 is on yyc-VirtualBox
Process 20 of 80 is on yyc-VirtualBox
Process 52 of 80 is on yyc-VirtualBox
Process 70 of 80 is on yyc-VirtualBox
Process 46 of 80 is on yyc-VirtualBox
Process 7 of 80 is on yyc-VirtualBox
Process 61 of 80 is on yyc-VirtualBox
Process 28 of 80 is on yyc-VirtualBox
Process 79 of 80 is on yyc-VirtualBox
Process 37 of 80 is on yyc-VirtualBox
Process 31 of 80 is on yyc-VirtualBox
Process 38 of 80 is on yyc-VirtualBox
Process 44 of 80 is on yyc-VirtualBox
Process 22 of 80 is on yyc-VirtualBox
Process 71 of 80 is on yyc-VirtualBox
Process 39 of 80 is on yyc-VirtualBox
Process 14 of 80 is on yyc-VirtualBox
Process 62 of 80 is on yyc-VirtualBox
Process 47 of 80 is on yyc-VirtualBox
Process 54 of 80 is on yyc-VirtualBox
Process 76 of 80 is on yyc-VirtualBox
Process 78 of 80 is on yyc-VirtualBox
Process 68 of 80 is on yyc-VirtualBox
Process 21 of 80 is on yyc-VirtualBox
Process 60 of 80 is on yyc-VirtualBox
Process 63 of 80 is on yyc-VirtualBox
```

图 11 选取案例程序进行集群测试的结果

```

Process 24 of 80 is on yyc-VirtualBox
Process 26 of 80 is on yyc-VirtualBox
Process 0 of 80 is on yyc-VirtualBox
Process 59 of 80 is on yyc-VirtualBox
Process 40 of 80 is on yyc-VirtualBox
Process 66 of 80 is on yyc-VirtualBox
pi is approximately 3.1415926544231270, Error is 0.0000000008333338
wall clock time = 1.603930

```

图 12 选取案例程序进行集群测试的结果

如果运行结果和上述相同,表示可以在集群下使用 MPI 运行文件,同理也可以将 HPL.dat 和 xhpl 文件都添加到共享目录下,然后任意计算机通过如下命令进行集群的性能测试:

```
mpirun -np 8 -f ./mpi_config ./xhpl
```

在测试运行时,有一个问题就是每次需要执行文件时都需要在两个计算机上都将在同一个节点放到同一个目录下,这个操作是比较繁琐的,可以通过安装 NFS 使得两个文件夹进行共享目录,在该目录下任意节点进行添加或者删除文件,其他的计算机的目录下都会响应进行变化。首先每个计算机节点安装 NFS。

```
sudo apt-get install nfs-kernel-server
```

在两个节点中要选取一个作为服务器,其他的做客户端,在这里使用的是 node1 作为服务器, node2 作为客户端。node1 需要修改 exports 文件。

```

sudo vim /etc/exports

/home/mpi_share 192.168.1.3(rw,sync,no_root_squash,no_subtree_check)

/home/mpi_share 192.168.1.3(rw,sync,no_root_squash,no_subtree_check)

sudo /etc/init.d/nfs-kernel-server restart

```

在其他节点中需要进行挂载,在 node2 中运行

```
sudo mount -t nfs node1:/home/mpi_share /home/mpi/share
```

然后可以在/home/mpi/share 文件夹中添加文件观察是否可以共享

4. 优化（详细阐述如何进行优化）并进行测试

HPL.dat 文件中最重要的几个参数（主要优化这几个）：

- problem sizes (matrix dimension N)

of problems(N) 行

Ns 行

of problems 设置测试问题的组数（也就是测试几次），Ns 行根据 of problems 规定设置相应数量的 N 的值（也就是每次测试的规模大小）。

- 求解问题规模越大浮点处理性能越高,但规模越大,测试时占用内存也更大。在集群测试中,内存为所有测试的节点的总内存。

由于 HPL 对双精度 (D P) 元素的 $N \times N$ 数组执行计算，并且每个双精度元素需要 8 字节=大小（双），因此 N 的问题大小所消耗的内存为 $8N^2$ 。

因为一般要为其他进程预留部分内存，因此消耗内存一般控制在实际内存的 80%–90%，例如取内存的 85%，则 $8N^2 = \text{ram} \times 0.85$ ，可得出 N 取值公式：

$$N = \sqrt{2} \left\{ 0.85 \times \frac{\text{RAMsize (GiB)} \times 1024^3}{\text{sizeof(double)}} \right\}$$

- 比较理想的问题规模值应该是 NB 值的倍数。

实际操作中可根据具体情况分析，监测内存占用量，适当调整规模值。

● block size NB

of NBs 行

NBs 行

为提高整体性能，HPL 采用分块矩阵的算法，of NBs 行表示要设置几组分块矩阵， NBs 根据 of NBs 规定设置相应数量的值， NBs 取值和软硬件许多因素密切相关，根据具体测试不断调节。

$NB \times 8$ 一定是 CPU L2 Cache line(单位 kb) 的倍数

一般通过单节点或单 CPU 测试得到较好的 NB 值，选择 3 个左右较好的 NBs 值，再扩大规模验证这些选择。

● process grid ($P \times Q$)

of process grids ($P \times Q$) 行

P 行

Q 行

这三行和 CPU 核心数量及运行 hpl 的线程有关。of process grids 表示 P 行和 Q 要使用几组网格，该行数字为多少，则 P 行和 Q 行就要各自设置相应数量的数值。

P 和 Q 的取值：

- $P \times Q$ = 系统 CPU Process 数（关闭超线程情况下即为 cpu 核心数，集群测试中 $P \times Q$ 为集群总的 Process）。
- $P \leq Q$ ，即 P 的值尽量比 Q 取小一点结果更好。
- $P = 2^n$ ，即 P 取值为 2 的幂结果更好。

HPL 中 L 分解 的列向通信采用二元交换法 (Binary Exchange)，当列向处理器个数 P 为 2 的幂时，性能最优。

5.使用 Vtune 进行分析

VTune 可视化性能分析器的安装:

1、下载 VTune 安装包，并解压:

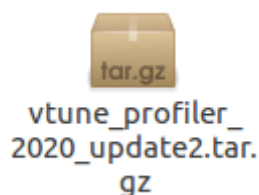


图 13 VTune 安装包

```
tar zxvf vtune_profiler_2020_update2.tar
```

进入解压后的文件夹，执行"install_GUI.sh"脚本，全部按照默认设置，根据安装向导安装即可。

```
sudo ./install_GUI.sh
```

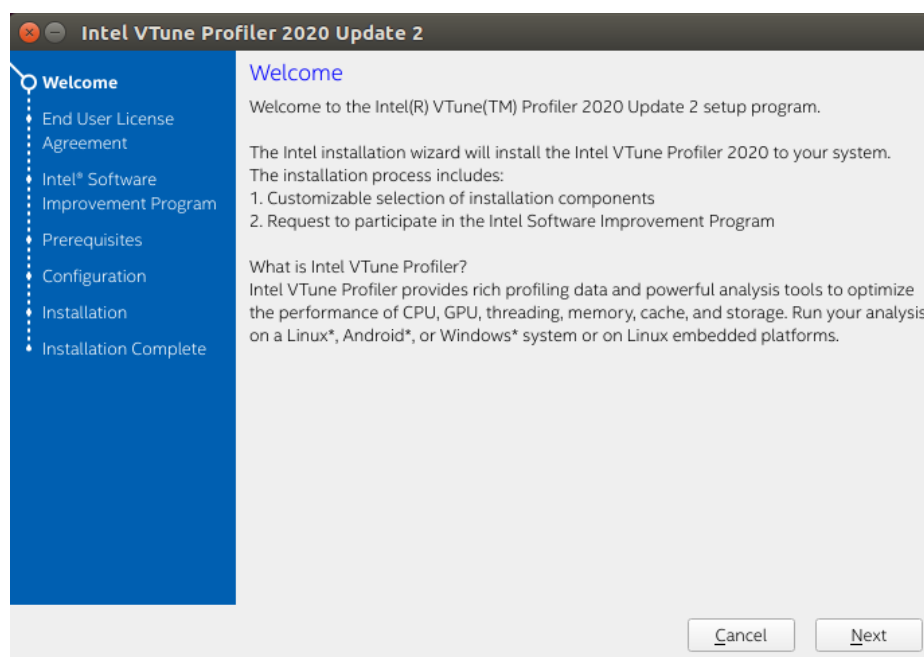


图 14 VTune 安装界面截图

安装完成后，需要先执行 VTune 安装成功后得到的文件:

```
cd /opt/intel/vtune_profiler_2020.2.0.610396
```

```
source ./vtune-vars.sh
```

```
vtune -collect hotspots mpirun -np 2 ./xhpl
```

运行结束后得到结果文件夹 r000hs，进入该文件夹，有如下文件:

```
root@yyc-VirtualBox:/opt/intel/vtune_profiler_2020.2.0.610396/r000hs# ls
archive  config  data.0  log     r000hs.vtune  sqlite-db
```

结果分析：使用 VTune-GUI 查看结果。

可以获得几类数据，分别为“Collection Log”、“Summary”、“Bottom-up”、“Caller/Callee”、“Top-down Tree” 和 “Platform”。

(1) 运行时间：主要有 CPU 运行总时间、有效时间、自旋时间（CPU 等待其它同步资源处理的自旋等待时间）、开销时间（花费在同步和线程库函数的时间）、暂停时间、总线程数量等信息。

Elapsed Time[?]: 3914.490s

- CPU Time[?]: 7116.860s
 - Effective Time[?]: 6772.195s
 - Spin Time[?]: 344.665s
 - Overhead Time[?]: 0s
- Total Thread Count: 7
- Paused Time[?]: 0s

(2) Top Hotspots 信息：列举 VTune 分析的程序里活跃度最高（最耗时）的部分，优化这些热点功能通常会提高整体应用程序性能。

我们可以看出 dgemm_kernel 这个代码段花费执行的时间最长：

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

| Function | Module | CPU Time [?] |
|--------------|--------------|-----------------------|
| dgemm_kernel | xhpl | 6202.313s |
| MPI_lprobe | libmpi.so.12 | 220.558s |
| HPL_lmul | xhpl | 200.831s |
| HPL_rand | xhpl | 111.767s |
| MPI_Send | libmpi.so.12 | 106.481s |
| [Others] | N/A* | 274.909s |

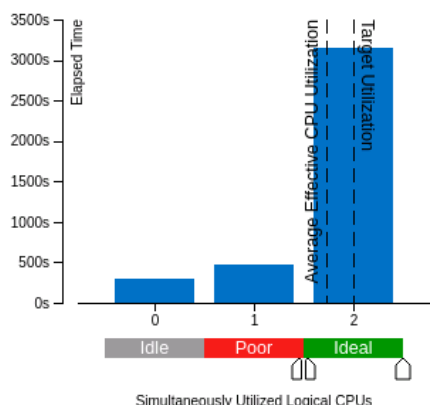
*N/A is applied to non-summable metrics.

(3) Effective CPU Utilization Histogram：有效 CPU 利用率直方图

这个柱状图显示了同时运行特定数量的 CPU 所占的运行时间，自旋时间和开销时间增加了空闲 CPU 的利用率。

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



(4) Collection and Platform Info: 包含了应用程序命令行、操作系统、CPU 等信息。

Collection and Platform Info

This section provides information about this collection, including result set size and collection platform data.

Application Command Line: `mpirun "-np" "2" ". /xhpl"`
Operating System: 3.10.0-1062.18.1.el7.x86_64 NAME="CentOS Linux" VERSION="7 (Core)" ID="centos"
ID_LIKE="rhel fedora" VERSION_ID="7" PRETTY_NAME="CentOS Linux 7 (Core)"
ANSI_COLOR="0;31" CPE_NAME="cpe:/o:centos:centos:7"
HOME_URL="https://www.centos.org/" BUG_REPORT_URL="https://bugs.centos.org/"
CENTOS_MANTISBT_PROJECT="CentOS-7"
CENTOS_MANTISBT_PROJECT_VERSION="7" REDHAT_SUPPORT_PRODUCT="centos"
REDHAT_SUPPORT_PRODUCT_VERSION="7"
Computer Name: iZxc9nkbdwojwrZ
Result Size: 144 MB
Collection start time: 12:56:59 06/05/2020 UTC
Collection stop time: 14:02:14 06/05/2020 UTC
Collector Type: User-mode sampling and tracing

Finalization mode: Fast. If the number of collected samples exceeds the threshold, this mode limits the number of processed samples to speed up post-processing.

⌵ CPU

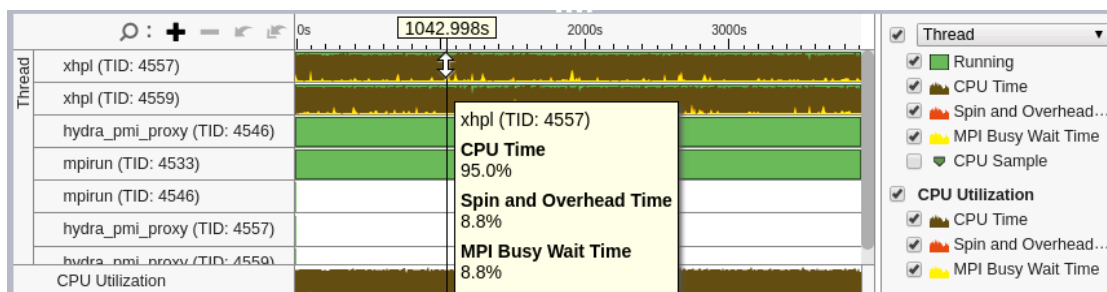
Name: Intel(R) Xeon(R) Processor code named Skylake
Frequency: 2.5 GHz
Logical CPU Count: 2

(5) Bottom-up: 查看函数/模块/线程调用时间的耗费，主要分析的数据有：进程、线程、模块、函数和调用的堆栈信息。可以显示程序的进程、线程号，函数的开始地址，CPU 开销时间，CPU 自旋时间等信息。

如图所示：前几个所占 CPU Time 多的函数是 Top Hotspots 显示的热点功能。

| Grouping: Function / Call Stack | | | | | |
|---------------------------------|-----------|--------------|-----------------|-------------|---------------|
| Function / Call Stack | CPU Time | Module | Function (Full) | Source File | Start Address |
| ▶ dgemm_kernel | 6202.313s | xhpl | dgemm_kernel | | 0x441200 |
| ▶ MPI_Iprobe | 220.558s | libmpi.so.12 | MPI_Iprobe | | 0x985f0 |
| ▶ HPL_lmul | 200.831s | xhpl | HPL_lmul | | 0x42b4d0 |
| ▶ HPL_rand | 111.767s | xhpl | HPL_rand | | 0x420100 |
| ▶ MPI_Send | 106.481s | libmpi.so.12 | MPI_Send | | 0x9f1f0 |
| ▶ dtrsm_kernel_LT | 53.541s | xhpl | dtrsm_kernel_LT | | 0x446a00 |
| ▶ HPL_dlaswp00N | 47.529s | xhpl | HPL_dlaswp00N | | 0x416550 |
| ▶ HPL_ladd | 39.217s | xhpl | HPL_ladd | | 0x42b460 |
| ▶ HPL_pdmatrixgen | 25.116s | xhpl | HPL_pdmatrixgen | | 0x411d30 |
| ▶ PMPI_Recv | 24.342s | libmpi.so.12 | PMPI_Recv | | 0x9c730 |
| ▶ HPL_setran | 22.897s | xhpl | HPL_setran | | 0x420180 |
| ▶ HPL_pdlange | 18.045s | xhpl | HPL_pdlange | | 0x40af60 |
| ▶ dgemm_itcopy | 15.604s | xhpl | dgemm_itcopy | | 0x442c00 |

下方可以看到不同线程各时刻 CPU Time、Spin and Overhead Time 和 MPI Busy Wait Time 的占比：



(6) Caller/Callee: 主要分析的数据：CPU 总利用时间、各个函数自我利用时间、各个函数的自我开销时间、各个函数的调用者和被调用者等。

| Function | CPU Time: Total | CPU Time: Self | Module | Function (Full) | Source File |
|-------------------|-----------------|----------------|--------------|-------------------|-------------|
| __libc_start_main | 100.0% | 0s | libc.so.6 | __libc_start_main | |
| _start | 100.0% | 0s | xhpl | _start | |
| [stack] | 100.0% | 0s | [stack] | [stack] | |
| main | 100.0% | 0.010s | xhpl | main | |
| HPL_pdtest | 100.0% | 0s | xhpl | HPL_pdtest | |
| HPL_pdgesv0 | 93.8% | 0.310s | xhpl | HPL_pdgesv0 | |
| HPL_pdgesv | 93.8% | 0s | xhpl | HPL_pdgesv | |
| HPL_pdupdatTT | 88.0% | 0.020s | xhpl | HPL_pdupdatTT | |
| cblas_dgemm | 87.4% | 0.020s | xhpl | cblas_dgemm | |
| dgemm_kernel | 87.1% | 6202.313s | xhpl | dgemm_kernel | |
| dgemm_nn | 86.5% | 0.135s | xhpl | dgemm_nn | |
| HPL_pdmatrixgen | 5.6% | 25.116s | xhpl | HPL_pdmatrixgen | |
| HPL_rand | 5.2% | 111.767s | xhpl | HPL_rand | |
| HPL_bcast_1ring | 4.8% | 1.494s | xhpl | HPL_bcast_1ring | |
| HPL_setran | 3.7% | 22.897s | xhpl | HPL_setran | |
| MPI_Iprobe | 3.1% | 220.558s | libmpi.so.12 | MPI_Iprobe | |
| HPL_lmul | 2.8% | 200.831s | xhpl | HPL_lmul | |
| MPI_Send | 1.5% | 106.481s | libmpi.so.12 | MPI_Send | |
| HPL_pdfact | 1.0% | 0.010s | xhpl | HPL_pdfact | |

(7) Top-down Tree: 以树形结构展示每个调用所花费的时间及占比，可以从时间花费最多的地方往下一层一层的展开，找到关键函数分析其性能。其分析的内容和 Caller/Callee 基本相同。

| Function Stack ▲ | CPU Time: Total [20] | CPU Time: Self [20] | Module | Function (Full) | Source File | Start Address |
|-----------------------|----------------------|---------------------|-----------|------------------|-------------|---------------|
| ▼ Total | 100.0% | 0s | | | | |
| ▼ [stack] | 100.0% | 0s | [stack] | [stack] | | 0 |
| ▼ _start | 100.0% | 0s | xhpl | _start | | 0x4025b2 |
| ▼ __libc_start_main | 100.0% | 0s | libc.so.6 | __libc_start_... | | 0x22410 |
| ▶ [Unknown stack fram | 0.0% | 0s | | [Unknown st... | | 0 |
| ▶ __libc_csu_init | 0.0% | 0s | xhpl | __libc_csu_init | | 0x457140 |
| ▶ main | 100.0% | 0.010s | xhpl | main | | 0x401c20 |
| ▼ _start | 0.0% | 0s | hydra_... | _start | | 0x403eeb |
| ▼ __libc_start_main | 0.0% | 0s | libc.so.6 | __libc_start_... | | 0x22410 |
| ▶ main | 0.0% | 0s | hydra_... | main | pmip.c | 0x402e40 |

四、实验结果与分析

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。

我们小组成员分别测试了自己的电脑性能并进行性能比较：

赵云飞

| 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 | 4.4928e-01 |
| 2 | 35 | 4 | 1 | 2 | 0.00 | 3.1121e-01 |
| 4 | 35 | 4 | 1 | 4 | 0.17 | 2.0150e-04 |
| 8 | 35 | 4 | 1 | 8 | 0.91 | 3.9709e-05 |
| 16 | 35 | 4 | 1 | 16 | 3.67 | 8.8034e-06 |
| ... | ... | ... | ... | ... | ... | ... |

任世奇

| 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 | 2.4515e-01 |
| 2 | 35 | 4 | 1 | 1 | 0.00 | 1.4939e-01 |
| 2 | 35 | 4 | 1 | 2 | 0.00 | 1.0172e-01 |
| 4 | 35 | 4 | 1 | 4 | 0.10 | 2.9482e-04 |
| 4 | 35 | 4 | 4 | 1 | 0.30 | 1.0038e-04 |
| 8 | 35 | 4 | 1 | 4 | 0.16 | 1.8774e-04 |

| | | | | | | |
|-----|-----|-----|-----|-----|------|------------|
| 8 | 35 | 4 | 1 | 8 | 0.23 | 1.3096e-04 |
| 8 | 35 | 4 | 4 | 1 | 0.97 | 3.1357e-05 |
| 8 | 35 | 4 | 8 | 1 | 1.30 | 2.3361e-05 |
| 16 | 35 | 4 | 1 | 4 | 0.51 | 5.9420e-05 |
| 16 | 35 | 4 | 1 | 16 | 0.65 | 4.6486e-05 |
| ... | ... | ... | ... | ... | ... | ... |

于永超

| 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 | 4.4613e-01 |
| 2 | 35 | 4 | 1 | 1 | 0.00 | 3.1740e-01 |
| 2 | 35 | 4 | 1 | 2 | 0.09 | 3.5640e-04 |
| 4 | 35 | 4 | 1 | 4 | 0.17 | 1.8011e-04 |
| 4 | 35 | 4 | 4 | 1 | 0.88 | 3.4476e-05 |
| 8 | 35 | 4 | 1 | 4 | 0.21 | 1.4396e-04 |
| 8 | 35 | 4 | 1 | 8 | 0.31 | 9.7367e-05 |
| 8 | 35 | 4 | 4 | 1 | 1.65 | 1.8438e-05 |
| 8 | 35 | 4 | 8 | 1 | 2.38 | 1.2772e-05 |
| 16 | 35 | 4 | 1 | 4 | 0.47 | 6.4528e-05 |
| 16 | 35 | 4 | 1 | 16 | 0.94 | 3.2360e-05 |
| ... | ... | ... | ... | ... | ... | ... |

王丹琳

| 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 | 3.0746e-01 |
| 2 | 35 | 4 | 1 | 1 | 0.00 | 3.0598e-01 |
| 4 | 35 | 4 | 1 | 1 | 0.00 | 2.9198e-01 |

| | | | | | | |
|-----|-----|-----|-----|-----|------|------------|
| 4 | 35 | 4 | 4 | 1 | 0.88 | 3.4575e-05 |
| 8 | 35 | 4 | 4 | 1 | 1.76 | 1.7323e-05 |
| 16 | 35 | 4 | 4 | 1 | 3.68 | 8.2669e-06 |
| ... | ... | ... | ... | ... | ... | ... |

根据表中结果，我们可发现，当其他条件不改变，使 $P > Q$ 时，执行时间会变长，结果会差一些； $P * Q$ 在 4 以下时 Gflops 较高；N 和 NB 在较大时 Gflops 较高，P 较小时 Gflops 较高； $P = 2^n$ ，即 P 取值为 2 的幂结果会更好。

2. 将组内成员的电脑构建为小“集群”，测试该“集群”的性能:

在进行测试时，两台计算机 CPU 和内存资源都会被一定量的占用，和两台计算机单独运行的表现差异不大，但是需要占用相对较多的网络资源。在同一个局域网下，其他设备的网络速度会降低很多。下图分别为运行情况下，node1 和 node2 资源使用情况。

| 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 | 12000 | 128 | 1 | 1 | 97.48 | 1.1820e+01 |
| 1 | 12000 | 192 | 1 | 1 | 96.50 | 1.1939e+01 |
| 1 | 12000 | 256 | 1 | 1 | 95.67 | 1.2044e+01 |
| 1 | 35 | 4 | 1 | 1 | 0.00 | 1.9016e-01 |
| 2 | 35 | 4 | 1 | 2 | 0.07 | 4.2133e-04 |
| 4 | 35 | 4 | 1 | 4 | 0.17 | 1.8121e-04 |
| 8 | 35 | 4 | 1 | 8 | 0.34 | 9.0008e-05 |
| 16 | 35 | 4 | 1 | 16 | 0.52 | 5.8927e-05 |
| 8 | 35 | 4 | 1 | 4 | 0.24 | 1.2473e-04 |
| 8 | 35 | 4 | 4 | 1 | 1.92 | 1.5804e-05 |
| 8 | 35 | 4 | 8 | 1 | 2.38 | 1.2775e-05 |
| 1 | 14142 | 128 | 1 | 1 | 156.71 | 1.2034e+01 |
| 1 | 14142 | 192 | 1 | 1 | 160.03 | 1.1784e+01 |
| 单机运行 | | | | | | |
| Node1 | | | | | | |
| 1 | 14142 | 192 | 1 | 1 | 149.95 | 1.2577e+01 |

| | | | | | | |
|-------|-------|-----|---|---|--------|------------|
| 1 | 12000 | 192 | 1 | 1 | 91.74 | 1.2560e+01 |
| Node2 | | | | | | |
| 1 | 14142 | 192 | 1 | 1 | 237.08 | 7.9546e+00 |
| 1 | 12000 | 192 | 1 | 1 | 145.89 | 7.8976e+00 |

实验发现，效果不如 node1 进行单机运行，初步考虑可能是由于网络的问题导致运行速度不增反减。

3. 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。

调整相关参数

1) 修改 N:

通过学习得知，N 矩阵的规模越大，有效计算所占的比例也越大，系统浮点处理性能越高；占用系统总内存的 80% 左右为最佳。使用公式 $N*N = \text{系统内存} * 80\%$

查看系统内存（以 byte）：

```
free -b
```

2) 修改 NB:

| Processor | NB |
|--|-----|
| Intel® Xeon® Processor X56*/E56*/E7-*/E7*/X7* (codenamed Nehalem or Westmere) | 256 |
| Intel Xeon Processor E26*/E26* v2 (codenamed Sandy Bridge or Ivy Bridge) | 256 |
| Intel Xeon Processor E26* v3/E26* v4 (codenamed Haswell or Broadwell) | 192 |
| Intel® Core™ i3/i5/i7-6* Processor (codenamed Skylake Client) | 192 |
| Intel® Xeon Phi™ Processor 72* (codenamed Knights Landing) | 336 |
| Intel Xeon Processor supporting Intel® Advanced Vector Extensions 512 (Intel® AVX-512) instructions (codenamed Skylake Server) | 384 |

图 8 NB 官网参考数据

NB 不可能太大或太小，一般在 256 以下； $NB \times 8$ 一定是 Cache line 的倍数，可以看到 cache line 是 64，一般取 128、192、256

查看 cache line

```
cat coherency_line_size
```

3) 修改 P、Q:

$P \times Q = \text{系统 CPU 数} = \text{进程数}$

根据资料得知, $P \times Q$ 在 4 以下时 Gflops 较高, 进程数取本机系统 cpu 数

查看本机 cpu 型号和个数以及 cache line 大小可以使用:

查看型号和个数

```
grep name /proc/cpuinfo
```

查看个数

```
grep 'physical id' /proc/cpuinfo
```

王丹琳:

```
Mem:      2090004480 1110777856 169672704 69824512 809553920 714280960
Swap:     1022357504 88080384 934277120
```

本机内存约为 $2e+9$, 根据 $N \times N \times 8 = \text{系统总内存} \times 80\%$ 计算, 取 $N=14142$

查看逻辑本机 cpu 型号与个数, $P \times Q$ 取 1

```
yyc@yyc-VirtualBox:/program_file/hpl-2.3/bin/name_str$ grep name /proc/cpuinfo
model name      : Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
```

NB 取 128 192 256

```
root@yyc-VirtualBox:~# cat /sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size
64
```

| 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 | 14142 | 128 | 1 | 1 | 91.43 | 1.2602e+01 |
| 1 | 14142 | 192 | 1 | 1 | 91.16 | 1.2639e+01 |
| 1 | 14142 | 256 | 1 | 1 | 91.58 | 1.2581e+01 |

【这里你们还要不要做啊?】

对调整相关参数性能变高的原因的分析:

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。但是, 矩阵规模 N 的增加达到一定阶段后性能却转而下降, 这是因为矩阵规模 N 的增加会导致内存消耗量的增加, 系统实际内存空间不足, 使用缓存, 性能会大幅度降低。所以, 要尽量增大矩阵规模 N 的同时, 又要保证不使用系统缓存。由于, 操作系统本身需要占用一定的内存, 除了矩阵 $A (N \times N)$ 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, 所以矩阵 A 占用系统总内存的 80% 左右为最佳, 即 $N \times N \times 8 = \text{系统总内存} \times 80\%$ 。我们通过这个式子得到了 N 的优化取值。

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，这里就有一个参数：NB。NB 的选择和软硬件许多因数密切相关，通过查阅资料，NB 不可能太大或太小，数据分块如果过大，则容易造成负载不平衡；但是如果数据分块过小，则通信开销就会很大，同样会影响计算的整体性能。综合以上考虑，要选取一个比较均衡的数值。一般在 256 以下，并且 $NB \times 8$ 一定是 Cache line 的倍数，同时我们也查阅官网，对官网提供的数据进行了参考。我们通过 `cat coherency_line_size` 命令来查看 Cache line 的大小，从而确定 NB 的合理取值。

对于 P, Q: 一个进程对应一个 CPU 可以得到最佳性能，所以 $P \times Q$ 选取系统 CPU 数时获得最好结果，即 $P \times Q = \text{系统 CPU 数} = \text{进程数}$ 。同时 $P \leq Q$ ，即 P 的值尽量取得小一点，列向通信量大于横向通信，结果会更优。

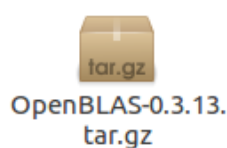
更换 BLAS

BLAS(basic linear algebra subroutine) 是基本线性代数子程序，是目前应用广泛的线性代数核心数学库。BLAS 最初是由 Fortran 语言编写，后来也出现了 C 语言版本的 cBLAS，其函数接口都大致相同，由于当前不同体系芯片厂商生成的处理器差异较大，在后序的发展中出现了两种 BLAS 设计方案：通用型和专用型。专用型 BLAS 一般由特定的芯片公司开发，针对自己平台的芯片特性进行代码优化，在自己平台上的性能往往相对较好，如 Intel 开发的 MKL，AMD 开发的 ACML，NVIDIA 开发的 cuBLAS；通用型 BLAS 一般由非营利性组织开发，在不同平台上都对开源代码 BLAS 进行适当的优化，目前主流的通用型 BLAS 是 ATLAS，GotoBLAS，BLIS 和 OpenBLAS。

所以，我们想通过换库来尝试实现性能的优化，我们使用基于 GotoBLAS 优化的 OpenBLAS 来处理矩阵之间的运算。

以下为安装、配置的具体步骤：

下载安装包



使用如下命令解压编译

```
tar zxvf OpenBLAS-0.3.13.tar.gz  
make PREFIX=/home/yye/ustb/openblas install
```

执行命令，修改

```
vim Make.name_str
```

按照如下进行修改

```
SHELL = /bin/sh
```

```
CD          = cd
CP          = cp
LN_S        = ln -s
MKDIR       = mkdir
RM          = /bin/rm -f
TOUCH       = touch
ARCH        = name_str
TOPdir      = /program_file/ hpl-2.3
INCdir      = $(TOPdir)/include
BINdir      = $(TOPdir)/bin/$(ARCH)
LIBdir      = $(TOPdir)/lib/$(ARCH)
HPLlib      = $(LIBdir)/libhpl.a
MPdir       = /program_file/ mpich-3.3.2
MPinc       = -I$(MPdir)/include
MPlib       = $(MPdir)/lib/libmpich.so
LAdir       = /program_file/ OpenBLAS
LAinc       = -I$(LAdir)/include
LAlib       = $(LAdir)/libopenblas.a
F2CDEFS     =
HPL_INCLUDES = -I$(INCdir) -I$(INCdir)/$(ARCH) $(LAinc) $(MPinc) -lpthread
HPL_LIBS     = $(HPLlib) $(LAlib) $(MPlib) -lpthread
HPL_OPTS     = -DHPL_CALL_CBLAS
HPL_DEFS     = $(F2CDEFS) $(HPL_OPTS) $(HPL_INCLUDES)
CC           = $(MPdir)/bin/mpicc -lpthread
CCNOOPT      = $(HPL_DEFS)
CCFLAGS      = $(HPL_DEFS) -fomit-frame-pointer -O3 -funroll-loops
LINKER       = $(MPdir)/bin/mpif77
LINKFLAGS    = $(CCFLAGS)
ARCHIVER     = ar
ARFLAGS      = r
```



```
RANLIB      = echo
```

修改 HPL_dat 的 PFACTS 与 RFACTS，尝试不同的组合，寻找 HPL 测试所得的实际峰值速度最优的组合：

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

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

测试 NBMINs 为 2 或者 4 的情况，选择最优的 NBMINs：

```
2  NBMINs
```

最后，执行命令

```
sudo ./xhpl
```

查看优化结果

| 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 | 14142 | 192 | 1 | 1 | 43.09 | 2.6743e+01 |

对比之前的结果，可以发现 HPL 测试所得的实际峰值速度得到了大幅度的提升，所以更换 BLAS 也可以实现 HPL 的优化。

五、实验心得体会

本次实验一共用时一个多星期，在安装配置中遇到了很多的问题，但是通过小组合作，都顺利解决了。通过本次实验，我们接触了计算机性能测试与优化，对 hpl 优化有了粗浅的认识。

在实验中我们熟悉了 Linux 系统的操作，包括虚拟机\双系统安装、Linux 系统配置、系统操作、软件安装、软件配置、系统属性查看等。我们学习了 MPI 的安装、配置、工作原理、使用方法等，了解了 GotoBLAS 的安装、配置、主要内容等，还学习了 HPL 的安装、配置。在主要实验过程中，我们着重学习了 HPL 的工作原理、文档说明、使用方法、调试方法。在实验过程中，我们对针对计算机属性的性能优化测试方法有了初步的认识，更加深入了解了计算机的 CPU、进程、内存的工作原理；我们还学习了 Vtune 可视化工具的安装、配置，以及使用 Vtune 进行性能分析的方法；我们还成功实现了在一个局域网内搭建集群，进行配置和调试。

本次的课程设计实践环节,我们通过小组分工通力合作,完成了相关的任务。这次的课程设计不仅帮助我们接触到了计算机性能测试,更让我们对计算机性能优化有了直观的认识。总之,在组内成员的通力配合之下我们完成了实验,收获颇丰!

六、参考文献

- [1] 金能智,文洮,杨博超,安文婷. 基于 HPL 的 Linux 高性能计算集群基准测试研究[J]. 现代信息科技, 2019, 3(14):60-62.
- [2] 贾迅, 邬贵明, 谢向辉. 异构高性能计算系统 Linpack 效率受限因素分析[J]. 计算机工程与科学, 2018, 40(02):224-230.
- [3] 马健. 龙芯高性能计算机软件系统的优化研究[D]. 中国科学院大学(中国科学院工程管理与信息技术学院), 2017.
- [4] 刘刚, 张恒, 张滇, 毛睿. 基于龙芯 3B 处理器的 Linpack 优化实现[J]. 深圳大学学报(理工版), 2014, 31(03):286-292.
- [5] 肖明旺, 许坚, 车永刚, 王正华. 一个实用高性能 PC 集群的 Linpack 测试与分析[J]. 计算机应用研究, 2004(09):183-184+187.
- [6] 张文力, 陈明宇, 樊建平. HPL 测试性能仿真与预测[J]. 计算机研究与发展, 2006(03):557-562.
- [7] 李铮. 基于 Linux 的小型高性能集群的研究和优化[D]. 上海交通大学, 2012.
- [8] 李铮, 薛质. 基于 Linux 的高性能集群的构建和性能优化[J]. 信息技术, 2012, 36(03):52-55.

附 1:

北京科技大学实验报告

学院: 计算机与通信工程学院

专业: 计算机科学与技术

班级: 计科 184

姓名: 王丹琳

学号: 41824179

实验日期: 2021 年 4 月 23 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 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 N 元一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。

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的基础上，可以调节问题规模大小 N (矩阵大小)、使用到的 CPU 数目、使用各种优化方法来执行该测试程序，以获取最佳的性能。HPL 采用高斯消元法求解线性方程组。当求解问题规模为 N 时，浮点运算次数为 $(2/3 * N^3 - 2 * N^2)$ 。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) 给出。

实验内容与步骤：

5. 安装 MPI

(5) 下载 MPI 安装包并解压

在官网 <http://www.mpich.org/downloads/> 下载 mpich 的最新版本(本次使用的是 3.3.2)。

在下载安装包目录下可以找到已经下载的 mpich-3.3.2.tar.gz，在该目录下打开终端使用下面命令解压压缩包：

```
tar -zxvf mpich-3.3.2.tar.gz
```

进入压缩目录下：

```
cd mpich-3.3.2
```

(6) 配置编译环境并进行安装

```
./configure --prefix=/program_file/ mpich-3.3.2
```

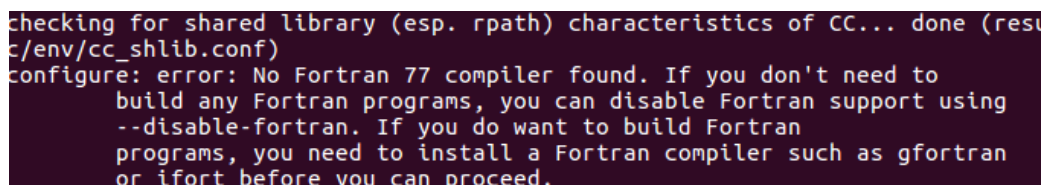
其中 /program_file/ mpich-3.3.2 为安装的路径，可以自由选择路径，该路径需要在下面的 HPL 配置中用到，这里需要将其记忆。

接下来进行安装，命令如下：

```
sudo make
```

```
sudo make install
```

Ubuntu 环境会遇到如下问题：



```
checking for shared library (esp. rpath) characteristics of CC... done (result stored in /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.
```

图 1 执行安装命令所遇到的问题

这里可能是因为缺少安装 C、C++、F77 和 F90 编译器，所以如果遇到以上问题，可以先进行如下命令。

对于 Ubuntu 可以使用下面命令进行补充安装

```
sudo apt-get install fort77  
sudo apt-get install gfortran
```

对于 Centos 可以使用下面命令进行补充安装。

```
yum install gcc -y  
yum install gcc-c++ -y  
yum install gcc-gfortran -y
```

再执行上面的安装命令，完成 MPI 的安装

```
sudo make  
sudo make install
```

（7）环境配置

使用 vi 或者 vim 打开 bashrc 文件，添加内容

```
vim ~/.bashrc
```

在最后一行添加如下内容（按 i 可以对 vim 进行编辑，编辑完后，按 esc 键再输入:wq 退出，vim 的学习链接 <http://www.runoob.com/linux/linux-vim.html>）

```
export PATH=/program_file/ mpich-3.3.2/bin:$PATH  
export LD_LIBRARY_PATH=/program_file/ mpich-3.3.2/lib:$LD_LIBRARY_PATH
```

编辑完成后输入下面内容激活环境变量

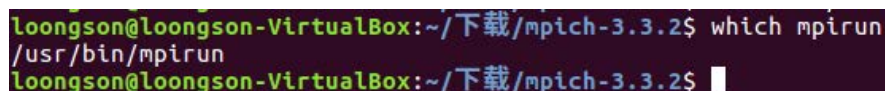
```
source ~/.bashrc
```

（8）测试 MPI 是否安装成功

检测命令导出是否成功

```
which mpirun
```

如果显示出安装路径（如下）则说明安装成功



```
loongson@loongson-VirtualBox:~/下载/mpich-3.3.2$ which mpirun  
/usr/bin/mpirun  
loongson@loongson-VirtualBox:~/下载/mpich-3.3.2$
```

图 2 MPI 安装成功的验证

在下载压缩包中有提供测试的样例代码，进入到压缩包路径下执行下面命令

```
cd examples  
mpicc hellow.c -o hellow
```

```
mpirun -np 4 ./hellow
```

如果运行结果如下证明安装已经完成。

```
loongson@loongson-VirtualBox:~/下载/mpich-3.3.2/examples$ mpirun -np 4 ./hellow
Hello world from process 2 of 4
Hello world from process 1 of 4
Hello world from process 0 of 4
Hello world from process 3 of 4
```

图 3 4 线程 hellow 程序运行结果

6. 配置 HPL

(3) 安装 GotoBLAS2

首先在官网下载（本次使用的是 GotoBLAS2-1.1.3），官网地址：

<https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2>

首先对压缩文件进行解压并进入该目录：

```
tar -xzf GotoBLAS2-1.13.tar.gz
```

```
cd GotoBLAS2-1.13
```

进行快速安装：

```
./quickbuild.64bit
```

如果出现大量 kernel/x86_64/gemm_ncopy_4.S 的错误（如下图）

```
../kernel/x86_64/gemm_ncopy_4.S: Assembler messages:
../kernel/x86_64/gemm_ncopy_4.S:175: 错误: invalid operands (*UND* and *ABS* sections) for '*'
../kernel/x86_64/gemm_ncopy_4.S:176: 错误: invalid operands (*UND* and *ABS* sections) for '*'
../kernel/x86_64/gemm_ncopy_4.S:177: 错误: invalid operands (*UND* and *ABS* sections) for '*'
../kernel/x86_64/gemm_ncopy_4.S:178: 错误: invalid operands (*UND* and *ABS* sections) for '*'
../kernel/x86_64/gemm_ncopy_4.S:180: 错误: invalid operands (*UND* and *ABS* sections) for '*'
../kernel/x86_64/gemm_ncopy_4.S:328: 错误: invalid operands (*UND* and *ABS* sections) for '*'
../kernel/x86_64/gemm_ncopy_4.S:329: 错误: invalid operands (*UND* and *ABS* sections) for '*'
../kernel/x86_64/gemm_ncopy_4.S:331: 错误: invalid operands (*UND* and *ABS* sections) for '*'
Makefile.L3:353: recipe for target 'sgemm_ncpy.o' failed
make[1]: *** [sgemm_ncpy.o] Error 1
make[1]: *** 正在等待未完成任务....
make[1]: Leaving directory '/home/loongson/下载/GotoBLAS2-1.13/kernel'
Makefile:89: recipe for target 'libs' failed
make: *** [libs] Error 1
```

图 4 GotoBLAS2 安装错误的截图

则需要添加编译选项（这个需要根据个人计算机 NEHALEM 是 Intel 的）：

```
make clean
```

```
make CC=gcc BINARY=64 TARGET=NEHALEM
```

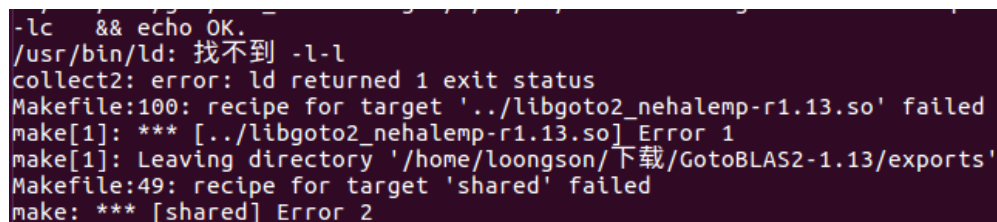
如果再遇到类似如下报错

```
/usr/lib64/gcc/x86_64-suse-linux/4.7/../../../../x86_64-suse-linux/bin/ld: cannot find -l-l
collect2: error: ld returned 1 exit status

make[1]: * [../libgoto2_nehalemp-r1.13.so] Error 1

make[1]: Leaving directory `/home/ken/bin/build/GotoBLAS2/exports' make:

[shared] Error 2
```



```
-lc && echo OK.
/usr/bin/ld: 找不到 -l-l
collect2: error: ld returned 1 exit status
Makefile:100: recipe for target '../libgoto2_nehalemp-r1.13.so' failed
make[1]: *** [../libgoto2_nehalemp-r1.13.so] Error 1
make[1]: Leaving directory '/home/loongson/下载/GotoBLAS2-1.13/exports'
Makefile:49: recipe for target 'shared' failed
make: *** [shared] Error 2
```

图 5 GotoBLAS2 安装错误的截图

则需要修改 f_check: 进入 f_check

```
vim f_check
```

修改第 298 行（如下图）：

```
print MAKEFILE "FEXTRALIB=$linker_L -lgfortran -lm -lquadmath -lm $linker_a\n";
```

再进行编译

```
make clean
```

```
make CC=gcc BINARY=64 TARGET=NEHALEM
```

（4）配置 HPL

首先要下载 hpl 的安装包（本次使用 hpl-2.3），官方网址：

<http://www.netlib.org/benchmark/hpl/hpl-2.3.tar.gz>

将下载内容进行解压

```
tar -xzf hpl-2.3.tar.gz
```

```
cd hpl-2.3
```

编辑 Make 配置文件

```
vim Make.name_str
```

需要编写的内容如下，标红色是有可能需要改动的内容

TOPdir 是 hpl 解压缩后所在的路径

MPdir 是 MPI 的安装路径

LAdir 是 GotoBLAS2 解压缩后所在路径

```
SHELL      = /bin/sh
CD          = cd
CP          = cp
LN_S        = ln -s
MKDIR       = mkdir
RM          = /bin/rm -f
TOUCH       = touch
ARCH        = name_str
TOPdir      = /program_file/ hpl-2.3
INCdir      = $(TOPdir)/include
BINDir      = $(TOPdir)/bin/$(ARCH)
LIBdir      = $(TOPdir)/lib/$(ARCH)
HPLlib      = $(LIBdir)/libhpl.a
MPdir       = /program_file/ mpich-3.3.2
MPinc       = -I$(MPdir)/include
MPLib       = $(MPdir)/lib/libmpich.so
LAdir       = /program_file/ GotoBLAS2
LAinc       =
LAlib       = $(LAdir)/libgoto2.a
F2CDEFS     =
HPL_INCLUDES = -I$(INCdir) -I$(INCdir)/$(ARCH) $(LAinc) $(MPinc) -lpthread
HPL_LIBS     = $(HPLlib) $(LAlib) $(MPLib) -lpthread
HPL_OPTS     = -DHPL_CALL_CBLAS
HPL_DEFS     = $(F2CDEFS) $(HPL_OPTS) $(HPL_INCLUDES)
CC           = $(MPdir)/bin/mpicc -lpthread
CCNOOPT      = $(HPL_DEFS)
CCFLAGS      = $(HPL_DEFS) -fomit-frame-pointer -O3 -funroll-loops
LINKER       = $(MPdir)/bin/mpif77
LINKFLAGS    = $(CCFLAGS)
```



```
ARCHIVER      = ar
ARFLAGS       = r
RANLIB        = echo
```

编写好后保存退出，执行：

```
make arch=name_str
```

如果配置文件没有问题并且，其他的两个已经安装成功执行完之后会出现 bin 目录，bin 目录中有 name_str 文件夹：

```
cd bin
cd name_str
ls
```

此时应该会看到两个文件 HPL.dat, xhpl



图 6 make 结果

进行初步尝试

```
mpirun -np 4 ./xhpl
```

如果运行正确会显示出类似下面的结果，

```
=====
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)= 1.88391027e-02 ..... PASSED
=====
T/V      N      NB      P      Q      Time      Gflops
-----
WR00R2R4 35      4      4      1      0.88      3.4575e-05
HPL_pdgesv() start time Sun Apr 25 09:36:33 2021
HPL_pdgesv() end time   Sun Apr 25 09:36:34 2021

=====
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)= 2.80968060e-02 ..... PASSED
=====
Finished 864 tests with the following results:
864 tests completed and passed residual checks,
0 tests completed and failed residual checks,
0 tests skipped because of illegal input values.
=====
End of Tests.
```

图 7 HPL 初次测试结果

7. 运行并行测试

1) 单机测试

修改 P*Q 的大小, 然后依次输入 `mpirun -np 4 ./xhpl`, `mpirun -np 8 ./xhpl`, `mpirun -np 16 ./xhpl` 进行测试, 并记录结果

2) 集群测试

将组内成员的电脑构建为小“集群”, 测试该“集群”的性能

本组实现的 MPI 集群搭建是两台计算机在同一个局域网下搭建的。如果没有公共的在同一个局域网下, 可以通过内网穿透然后再进行连接, 在搭建集群之前, 首先要保证, 两台计算机已经安装了 MPICH 并且版本最好一致。此外除了 MPICH 版本一致外, GCC 以及 fortran 的版本最好也要相同, 防止不能运行共同的文件, 最好的方法就是使用两个一样 Linux 版本的计算机测试。

在测试之前, 需要保证两个计算机的用户名是相同的, 不相同最好改一下用户名, 因为后面每个虚拟机节点之间免密登录时需要保证用户名一致, 这样可以避免出现未知的问题。

具体的安装方法:

1) 两台计算机安装 SSH, 并设置 SSH 免密登录

使用 `ifconfig` 命令查看两个计算机的 IP 地址, 修改两个计算机的 `hosts` 文件, 在两个计算机的 `hosts` 文件分别添加节点 IP。注意: 由于每个机器的 ip 会自动分配发生变化, 所以在在此之前, 我们已经完成了虚拟机的静态 ip 设置

```
sudo vim /etc/hosts
```

```
192.168.43.5 node1
```

```
192.168.43.7 node2
```

我们可以通过 `ping` 命令判断是否机器可以 `ping` 到, 来检查是否处于同一个局域网内, 若出现错误了, 需要检查 `hosts` 文件是否修改正确。

```
ping node1
```

```
ping node2
```

```
yyc@yyc-VirtualBox:~$ ping node2
PING node2 (192.168.43.7) 56(84) bytes of data.
64 bytes from node2 (192.168.43.7): icmp_seq=1 ttl=64 time=6.52 ms
64 bytes from node2 (192.168.43.7): icmp_seq=2 ttl=64 time=26.2 ms
64 bytes from node2 (192.168.43.7): icmp_seq=3 ttl=64 time=48.6 ms
64 bytes from node2 (192.168.43.7): icmp_seq=4 ttl=64 time=276 ms
64 bytes from node2 (192.168.43.7): icmp_seq=5 ttl=64 time=86.5 ms
64 bytes from node2 (192.168.43.7): icmp_seq=6 ttl=64 time=225 ms
64 bytes from node2 (192.168.43.7): icmp_seq=7 ttl=64 time=143 ms
64 bytes from node2 (192.168.43.7): icmp_seq=8 ttl=64 time=163 ms
64 bytes from node2 (192.168.43.7): icmp_seq=9 ttl=64 time=188 ms
64 bytes from node2 (192.168.43.7): icmp_seq=10 ttl=64 time=85.1 ms
64 bytes from node2 (192.168.43.7): icmp_seq=11 ttl=64 time=28.0 ms
64 bytes from node2 (192.168.43.7): icmp_seq=12 ttl=64 time=50.8 ms
```

图 8 node1 ping node2 结果

在两个节点分别安装 ssh:

```
sudo apt-get install ssh
```

此时，可以先尝试是否可以通过 `ssh` 命令进行各计算机的远程登录，若已经实行了免密登录则跳过下面几步，但是一般情况下，此时的登录是需要密码的。

ssh yyc@node1

ssh yyc@node2

```
yyc@yyc-VirtualBox:~$ ssh yyc@node2
The authenticity of host 'node2 (192.168.43.7)' can't be established.
ECDSA key fingerprint is SHA256:GvonwEYtZT5od00vdeXIWzJKhPQ95Rr+2ahL7doj0QC.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'node2,192.168.43.7' (ECDSA) to the list of known
ts.
yyc@node2's password:
Welcome to Ubuntu 16.04.7 LTS (GNU/Linux 4.15.0-142-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

28 个可升级软件包。
0 个安全更新。

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.
```

图 9 node1 通过 ssh 命令进行 node2 远程登录结果

生成 ssh 的公钥和私钥，并讲两个节点的公钥都添加到自己的 `authorized_keys` 中，以此实现免密登录了。

具体操作如下:

node1 , node 2 分别输入以下命令，生成公钥和私钥（在.ssh 文件中）

```
ssh-keygen -t rsa
```

```

yyc@yyc-VirtualBox:~$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/yyc/.ssh/id_rsa):
Created directory '/home/yyc/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/yyc/.ssh/id_rsa.
Your public key has been saved in /home/yyc/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:6d3KCWEuEGpL88z8P/V8aled0pa3iS2B6xQjVLj8YS0 yyc@yyc-VirtualBox
The key's randomart image is:
+----[RSA 2048]----+
|          ..        |
|         ..        |
|      .   . . .    |
|     . . .+ E .    |
|    = .   S.oo+   o |
| o B .   + oo+o. .o |
|   . = . +..+o +.  |
|    . . .O.++*=..  |
|    . . . =oo=+++. |
+-----[SHA256]-----+

```

图 9 node1 生成公钥和私钥

在 node1 与 node2 输入如下命令，完成对自己计算机的认证，该操作完成后，使用 ssh 可以对自己免密登录

```
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

node1 节点进行如下操作，将 node1 的公钥传到 node2

```
scp ~/.ssh/id_rsa.pub yyc@node2:~/.ssh/id_rsa.pub.node1
```

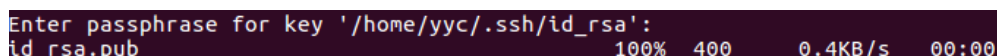


图 10 node1 的公钥传到 node2

node2 节点进行如下操作，将刚刚传到的 node1 公钥添加认证

```
cat ~/.ssh/id_rsa.pub.node1 >> ~/.ssh/authorized_keys
```

node2 节点进行如下操作，将 node1 的公钥传到 node1

```
scp ~/.ssh/id_rsa.pub 用户名@node1:~/.ssh/id_rsa.pub.node2
```

node1 节点进行如下操作，将刚刚传到的 node2 公钥添加认证

```
cat ~/.ssh/id_rsa.pub.node2 >> ~/.ssh/authorized_keys
```

再次使用 ssh 命令测试是否 node1 和 node2 之间都可以进行免密登录。

```
ssh yyc@node1
```

```
ssh yyc@node2
```

如果可以相互连接成功，则表示 node1 和 node2 之间通过 ssh 可以进行免密登录，那么就可以尝试集群共同运行程序进行尝试。

在测试之前，首先需要在两个计算机上创建一个共享的目录：在 node1 ,node2 分别创建 /home/mpi_share 文件夹，然后加入 cpi 程序，再同加入 mpi_config 文件，作为运行的配置文件，内容如下：

```
node1:4
```

```
node2:4
```

这里的添加根据自己的 CPU 核心数进行添加，然后在任意节点进行运行：

```
mpirun -n 80 -f ./mpi_config ./cpi1
```

运行结果如下：

```
root@yyc-VirtualBox:/mpi_share# mpirun -n 80 -f ./mpi_config ./cpi
Process 53 of 80 is on yyc-VirtualBox
Process 45 of 80 is on yyc-VirtualBox
Process 4 of 80 is on yyc-VirtualBox
Process 6 of 80 is on yyc-VirtualBox
Process 77 of 80 is on yyc-VirtualBox
Process 13 of 80 is on yyc-VirtualBox
Process 36 of 80 is on yyc-VirtualBox
Process 55 of 80 is on yyc-VirtualBox
Process 5 of 80 is on yyc-VirtualBox
Process 29 of 80 is on yyc-VirtualBox
Process 30 of 80 is on yyc-VirtualBox
Process 20 of 80 is on yyc-VirtualBox
Process 52 of 80 is on yyc-VirtualBox
Process 70 of 80 is on yyc-VirtualBox
Process 46 of 80 is on yyc-VirtualBox
Process 7 of 80 is on yyc-VirtualBox
Process 61 of 80 is on yyc-VirtualBox
Process 28 of 80 is on yyc-VirtualBox
Process 79 of 80 is on yyc-VirtualBox
Process 37 of 80 is on yyc-VirtualBox
Process 31 of 80 is on yyc-VirtualBox
Process 38 of 80 is on yyc-VirtualBox
Process 44 of 80 is on yyc-VirtualBox
Process 22 of 80 is on yyc-VirtualBox
Process 71 of 80 is on yyc-VirtualBox
Process 39 of 80 is on yyc-VirtualBox
Process 14 of 80 is on yyc-VirtualBox
Process 62 of 80 is on yyc-VirtualBox
Process 47 of 80 is on yyc-VirtualBox
Process 54 of 80 is on yyc-VirtualBox
Process 76 of 80 is on yyc-VirtualBox
Process 78 of 80 is on yyc-VirtualBox
Process 68 of 80 is on yyc-VirtualBox
Process 21 of 80 is on yyc-VirtualBox
Process 60 of 80 is on yyc-VirtualBox
Process 63 of 80 is on yyc-VirtualBox
```

图 11 选取案例程序进行集群测试的结果

```
Process 24 of 80 is on yyc-VirtualBox
Process 26 of 80 is on yyc-VirtualBox
Process 0 of 80 is on yyc-VirtualBox
Process 59 of 80 is on yyc-VirtualBox
Process 40 of 80 is on yyc-VirtualBox
Process 66 of 80 is on yyc-VirtualBox
pi is approximately 3.1415926544231270, Error is 0.000000008333338
wall clock time = 1.603930
```

图 12 选取案例程序进行集群测试的结果

如果运行结果和上述相同,表示可以在集群下使用 MPI 运行文件,同理也可以将 HPL.dat 和 xhpl 文件都添加到共享目录下,然后任意计算机通过如下命令进行集群的性能测试:

```
mpirun -np 8 -f ./mpi_config ./xhpl
```

在测试运行时,有一个问题就是每次需要执行文件时都需要在两个计算机上都将同一个节点放到同一个目录下,这个操作是比较繁琐的,可以通过安装 NFS 使得两个文件夹进行共享目录,在该目录下任意节点进行添加或者删除文件,其他的计算机的目录下都会响应进行变化。首先每个计算机节点安装 NFS。

```
sudo apt-get install nfs-kernel-server
```

在两个节点中要选取一个作为服务器,其他的做客户端,在这里使用的是 node1 作为服务器, node2 作为客户端。node1 需要修改 exports 文件。

```
sudo vim /etc/exports
```

```
/home/mpi_share 192.168.1.3(rw, sync, no_root_squash, no_subtree_check)
```

```
/home/mpi_share 192.168.1.3(rw, sync, no_root_squash, no_subtree_check)
```

```
sudo /etc/init.d/nfs-kernel-server restart
```

在其他节点中需要进行挂载，在 node2 中运行

```
sudo mount -t nfs node1:/home/mpi_share /home/mpi/share
```

然后可以在/home/mpi/share 文件夹中添加文件观察是否可以共享

8. 优化（详细阐述如何进行优化）并进行测试

HPL.dat 文件中最重要的几个参数（主要优化这几个）：

- problem sizes (matrix dimension N)

of problems(N) 行

Ns 行

of problems 设置测试问题的组数（也就是测试几次），Ns 行根据 of problems 规定设置相应数量的 N 的值（也就是每次测试的规模大小）。

- 求解问题规模越大浮点处理性能越高，但规模越大，测试时占用内存也更大。在集群测试中，内存为所有测试的节点的总内存。

由于 HPL 对双精度(D P)元素的 $N \times N$ 数组执行计算，并且每个双精度元素需要 8 字节=大小（双），因此 N 的问题大小所消耗的内存为 $8N^2$ 。

因为一般要为其他进程预留部分内存，因此消耗内存一般控制在实际内存的 80%–90%，例如取内存的 85%，则 $8N^2 = \text{ram} \times 0.85$ ，可得出 N 取值公式：

$$N = \sqrt[2]{0.85 \times \frac{\text{RAMsize}(\text{GiB}) \times 1024^3}{\text{sizeof}(\text{double})}}$$

- 比较理想的问题规模值应该是 NB 值的倍数。

实际操作中可根据具体情况分析，监测内存占用量，适当调整规模值。

- block size NB

of NBs 行

NBs 行

为提高整体性能，HPL 采用分块矩阵的算法，of NBs 行表示要设置几组分块矩阵，NBs 根据 ofNBs 规定设置相应数量的值，NBs 取值和软硬件许多因素密切相关，根据具体测试不断调节。

NB×8 一定是 CPU L2 Cache line(单位 kb) 的倍数

一般通过单节点或单 CPU 测试得到较好的 NB 值, 选择 3 个左右较好的 NBs 值, 再扩大规模验证这些选择。

- process grid (P x Q)

of process grids (P x Q) 行

P 行

Q 行

这三行和 CPU 核心数量及运行 hpl 的线程有关。of process grids 表示 P 行和 Q 要使用几组网格, 该行数字为多少, 则 P 行和 Q 行就要各自设置相应数量的数值。

P 和 Q 的取值:

- $P \times Q$ = 系统 CPU Process 数 (关闭超线程情况下即为 cpu 核心数, 集群测试中 $P \times Q$ 为集群总的 Process)。
- $P \leq Q$, 即 P 的值尽量比 Q 取小一点结果更好。
- $P = 2^n$, 即 P 取值为 2 的幂结果更好。

HPL 中 L 分解 的列向通信采用二元交换法 (Binary Exchange), 当列向处理器个数 P 为 2 的幂时, 性能最优。

5.使用 Vtune 进行分析

VTune 可视化性能分析器的安装:

1、下载 VTune 安装包, 并解压:

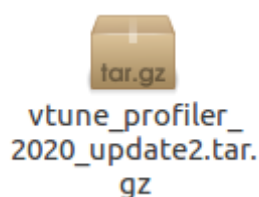


图 13 VTune 安装包

```
tar zxvf vtune_profiler_2020_update2.tar
```

进入解压后的文件夹, 执行 "install_GUI.sh" 脚本, 全部按照默认设置, 根据安装向导安装即可。

```
sudo ./install_GUI.sh
```

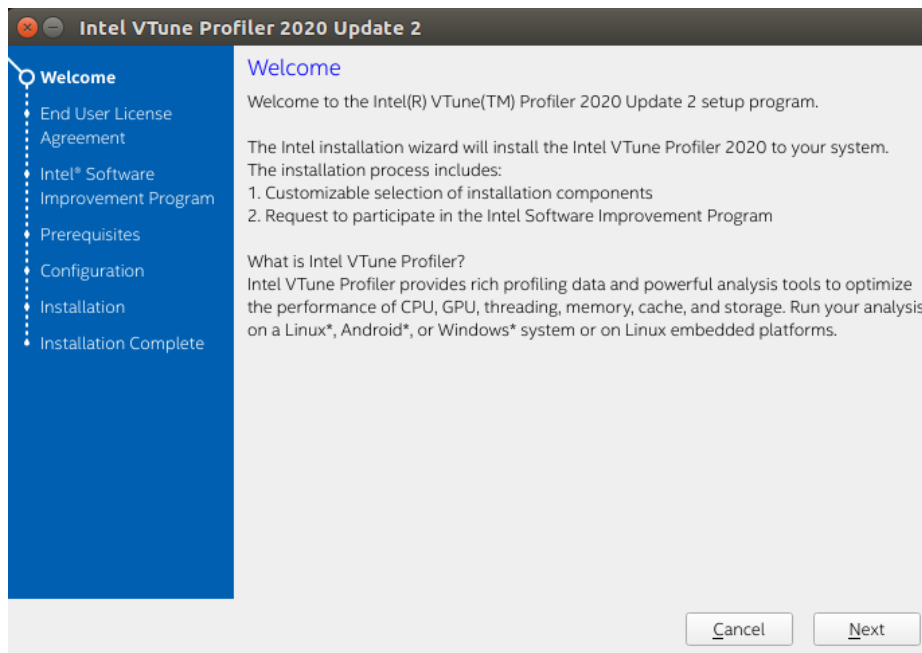



图 14 VTune 安装界面截图

安装完成后，需要先执行 VTune 安装成功后得到的文件：

```
cd /opt/intel/vtune_profiler_2020.2.0.610396
```

```
source ./vtune-vars.sh
```

```
vtune -collect hotspots mpirun -np 2 ./xhpl
```

运行结束后得到结果文件夹 r000hs，进入该文件夹，有如下文件：

```
root@yyc-VirtualBox:/opt/intel/vtune_profiler_2020.2.0.610396/r000hs# ls
archive config data.0 log r000hs.vtune sqlite-db
```

结果分析：使用 VTune-GUI 查看结果。

可以获得几类数据，分别为“Collection Log”、“Summary”、“Bottom-up”、“Caller/Callee”、“Top-down Tree”和“Platform”。

(1) 运行时间：主要有 CPU 运行总时间、有效时间、自旋时间（CPU 等待其它同步资源处理的自旋等待时间）、开销时间（花费在同步和线程库函数的时间）、暂停时间、总线线程数量等信息。

Elapsed Time [?]: 3914.490s

- CPU Time [?]: 7116.860s
- Effective Time [?]: 6772.195s
- Spin Time [?]: 344.665s
- Overhead Time [?]: 0s
- Total Thread Count: 7
- Paused Time [?]: 0s

(2) Top Hotspots 信息：列举 VTune 分析的程序里活跃度最高（最耗时）的部分，优化这些热点功能通常会提高整体应用程序性能。

我们可以看出 `dgemm_kernel` 这个代码段花费执行的时间最长：



Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

| Function | Module | CPU Time [?] |
|---------------------------|--------------|-----------------------|
| <code>dgemm_kernel</code> | xhpl | 6202.313s |
| <code>MPI_Iprobe</code> | libmpi.so.12 | 220.558s |
| <code>HPL_lmul</code> | xhpl | 200.831s |
| <code>HPL_rand</code> | xhpl | 111.767s |
| <code>MPI_Send</code> | libmpi.so.12 | 106.481s |
| [Others] | N/A* | 274.909s |

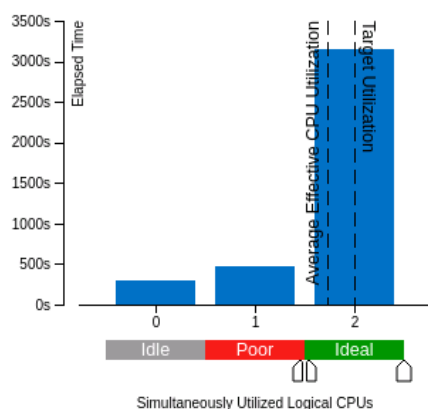
*N/A is applied to non-summable metrics.

(3) Effective CPU Utilization Histogram: 有效 CPU 利用率直方图

这个柱状图显示了同时运行特定数量的 CPU 所占的运行时间，自旋时间和开销时间增加了空闲 CPU 的利用率。

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



(4) Collection and Platform Info: 包含了应用程序命令行、操作系统、CPU 等信息。

Collection and Platform Info

This section provides information about this collection, including result set size and collection platform data.

Application Command Line: `mpirun "-np" "2" ".xhpl"`
 Operating System: 3.10.0-1062.18.1.el7.x86_64 NAME="CentOS Linux" VERSION="7 (Core)" ID="centos" ID_LIKE="rhel fedora" VERSION_ID="7" PRETTY_NAME="CentOS Linux 7 (Core)" ANSI_COLOR="0;31" CPE_NAME="cpe:/o:centos:centos:7" HOME_URL="https://www.centos.org/" BUG_REPORT_URL="https://bugs.centos.org/" CENTOS_MANTISBT_PROJECT="CentOS-7" CENTOS_MANTISBT_PROJECT_VERSION="7" REDHAT_SUPPORT_PRODUCT="centos" REDHAT_SUPPORT_PRODUCT_VERSION="7"
 Computer Name: iZxc9nkdbwojwrZ
 Result Size: 144 MB
 Collection start time: 12:56:59 06/05/2020 UTC
 Collection stop time: 14:02:14 06/05/2020 UTC
 Collector Type: User-mode sampling and tracing
Finalization mode: Fast. If the number of collected samples exceeds the threshold, this mode limits the number of processed samples to speed up post-processing.

CPU

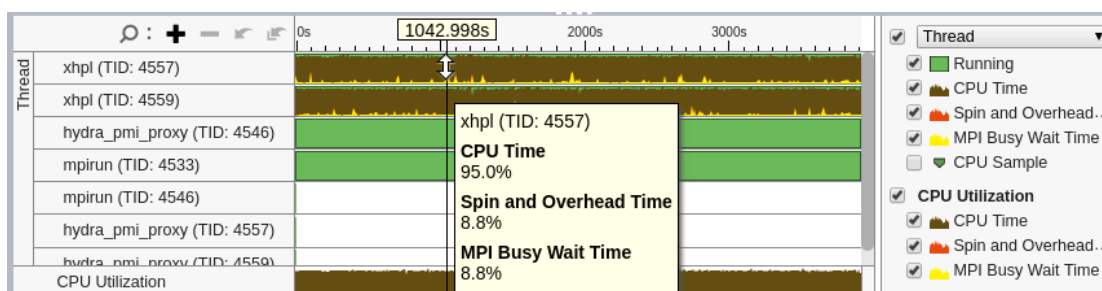
Name: Intel(R) Xeon(R) Processor code named Skylake
 Frequency: 2.5 GHz
 Logical CPU Count: 2

(5) **Bottom-up**: 查看函数/模块/线程调用时间的耗费，主要分析的数据有：进程、线程、模块、函数和调用的堆栈信息。可以显示程序的进程、线程号，函数的开始地址，CPU 开销时间，CPU 自旋时间等信息。

如图所示：前几个所占 CPU Time 多的函数是 Top Hotspots 显示的热点功能。

| Grouping: Function / Call Stack | | | | | |
|---------------------------------|-----------|--------------|-----------------|-------------|---------------|
| Function / Call Stack | CPU Time | Module | Function (Full) | Source File | Start Address |
| ▶ dgemm_kernel | 6202.313s | xhpl | dgemm_kernel | | 0x441200 |
| ▶ MPI_Iprobe | 220.558s | libmpi.so.12 | MPI_Iprobe | | 0x985f0 |
| ▶ HPL_lmul | 200.831s | xhpl | HPL_lmul | | 0x42b4d0 |
| ▶ HPL_rand | 111.767s | xhpl | HPL_rand | | 0x420100 |
| ▶ MPI_Send | 106.481s | libmpi.so.12 | MPI_Send | | 0x9f1f0 |
| ▶ dtrsm_kernel_LT | 53.541s | xhpl | dtrsm_kernel_LT | | 0x446a00 |
| ▶ HPL_dlaswp00N | 47.529s | xhpl | HPL_dlaswp00N | | 0x416550 |
| ▶ HPL_ladd | 39.217s | xhpl | HPL_ladd | | 0x42b460 |
| ▶ HPL_pdmagten | 25.116s | xhpl | HPL_pdmagten | | 0x411d30 |
| ▶ PMPI_Recv | 24.342s | libmpi.so.12 | PMPI_Recv | | 0x9c730 |
| ▶ HPL_setran | 22.897s | xhpl | HPL_setran | | 0x420180 |
| ▶ HPL_pdlange | 18.045s | xhpl | HPL_pdlange | | 0x40af60 |
| ▶ dgemm_itcopy | 15.604s | xhpl | dgemm_itcopy | | 0x442c00 |

下方可以看到不同线程各时刻 CPU Time、Spin and Overhead Time 和 MPI Busy Wait Time 的占比：



(6) Caller/Callee: 主要分析的数据: CPU 总利用时间、各个函数自我利用时间、各个函数的自我开销时间、各个函数的调用者和被调用者等。

| Function | CPU Time: Total ▾ | CPU Time: Self | Module | Function (Full) | Source File |
|-------------------|-------------------|----------------|--------------|-------------------|-------------|
| __libc_start_main | 100.0% | 0s | libc.so.6 | __libc_start_main | |
| _start | 100.0% | 0s | xhpl | _start | |
| [stack] | 100.0% | 0s | [stack] | [stack] | |
| main | 100.0% | 0.010s | xhpl | main | |
| HPL_pdtest | 100.0% | 0s | xhpl | HPL_pdtest | |
| HPL_pdgesv0 | 93.8% | 0.310s | xhpl | HPL_pdgesv0 | |
| HPL_pdgesv | 93.8% | 0s | xhpl | HPL_pdgesv | |
| HPL_pdupdateTT | 88.0% | 0.020s | xhpl | HPL_pdupdateTT | |
| cblas_dgemm | 87.4% | 0.020s | xhpl | cblas_dgemm | |
| dgemm_kernel | 87.1% | 6202.313s | xhpl | dgemm_kernel | |
| dgemm_nn | 86.5% | 0.135s | xhpl | dgemm_nn | |
| HPL_pdmatrixgen | 5.6% | 25.116s | xhpl | HPL_pdmatrixgen | |
| HPL_rand | 5.2% | 111.767s | xhpl | HPL_rand | |
| HPL_bcast_1ring | 4.8% | 1.494s | xhpl | HPL_bcast_1ring | |
| HPL_setran | 3.7% | 22.897s | xhpl | HPL_setran | |
| MPI_lprobe | 3.1% | 220.558s | libmpi.so.12 | MPI_lprobe | |
| HPL_lmul | 2.8% | 200.831s | xhpl | HPL_lmul | |
| MPI_Send | 1.5% | 106.481s | libmpi.so.12 | MPI_Send | |
| HPL_pdfact | 1.0% | 0.010s | xhpl | HPL_pdfact | |

(7) Top-down Tree: 以树形结构展示每个调用所花费的时间及占比, 可以从时间花费最多的地方往下一层一层的展开, 找到关键函数分析其性能。其分析的内容和 Caller/Callee 基本相同。

| Function Stack ▲ | CPU Time: Total | CPU Time: Self | Module | Function (Full) | Source File | Start Address |
|-------------------------|-----------------|----------------|-----------|-----------------|-------------|---------------|
| ▼ Total | 100.0% | 0s | | | | |
| ▼ [stack] | 100.0% | 0s | [stack] | [stack] | | 0 |
| ▼ _start | 100.0% | 0s | xhpl | _start | | 0x4025b2 |
| ▼ __libc_start_main | 100.0% | 0s | libc.so.6 | __libc_start... | | 0x22410 |
| ▶ [Unknown stack frame] | 0.0% | 0s | | [Unknown st... | | 0 |
| ▶ __libc_csu_init | 0.0% | 0s | xhpl | __libc_csu_init | | 0x457140 |
| ▶ main | 100.0% | 0.010s | xhpl | main | | 0x401c20 |
| ▼ _start | 0.0% | 0s | hydra... | _start | | 0x403eeb |
| ▼ __libc_start_main | 0.0% | 0s | libc.so.6 | __libc_start... | | 0x22410 |
| ▶ main | 0.0% | 0s | hydra... | main | pmip.c | 0x402e40 |

实验数据:

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。

我们小组成员分别测试了自己的电脑性能并进行性能比较:

赵云飞

| 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 | 4.4928e-01 |
| 2 | 35 | 4 | 1 | 2 | 0.00 | 3.1121e-01 |

| | | | | | | |
|-----|-----|-----|-----|-----|------|------------|
| 4 | 35 | 4 | 1 | 4 | 0.17 | 2.0150e-04 |
| 8 | 35 | 4 | 1 | 8 | 0.91 | 3.9709e-05 |
| 16 | 35 | 4 | 1 | 16 | 3.67 | 8.8034e-06 |
| ... | ... | ... | ... | ... | ... | ... |

任世奇

| 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 | 2.4515e-01 |
| 2 | 35 | 4 | 1 | 1 | 0.00 | 1.4939e-01 |
| 2 | 35 | 4 | 1 | 2 | 0.00 | 1.0172e-01 |
| 4 | 35 | 4 | 1 | 4 | 0.10 | 2.9482e-04 |
| 4 | 35 | 4 | 4 | 1 | 0.30 | 1.0038e-04 |
| 8 | 35 | 4 | 1 | 4 | 0.16 | 1.8774e-04 |
| 8 | 35 | 4 | 1 | 8 | 0.23 | 1.3096e-04 |
| 8 | 35 | 4 | 4 | 1 | 0.97 | 3.1357e-05 |
| 8 | 35 | 4 | 8 | 1 | 1.30 | 2.3361e-05 |
| 16 | 35 | 4 | 1 | 4 | 0.51 | 5.9420e-05 |
| 16 | 35 | 4 | 1 | 16 | 0.65 | 4.6486e-05 |
| ... | ... | ... | ... | ... | ... | ... |

于永超

| 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 | 4.4613e-01 |
| 2 | 35 | 4 | 1 | 1 | 0.00 | 3.1740e-01 |
| 2 | 35 | 4 | 1 | 2 | 0.09 | 3.5640e-04 |
| 4 | 35 | 4 | 1 | 4 | 0.17 | 1.8011e-04 |
| 4 | 35 | 4 | 4 | 1 | 0.88 | 3.4476e-05 |

| | | | | | | |
|-----|-----|-----|-----|-----|------|------------|
| 8 | 35 | 4 | 1 | 4 | 0.21 | 1.4396e-04 |
| 8 | 35 | 4 | 1 | 8 | 0.31 | 9.7367e-05 |
| 8 | 35 | 4 | 4 | 1 | 1.65 | 1.8438e-05 |
| 8 | 35 | 4 | 8 | 1 | 2.38 | 1.2772e-05 |
| 16 | 35 | 4 | 1 | 4 | 0.47 | 6.4528e-05 |
| 16 | 35 | 4 | 1 | 16 | 0.94 | 3.2360e-05 |
| ... | ... | ... | ... | ... | ... | ... |

王丹琳

| 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 | 3.0746e-01 |
| 2 | 35 | 4 | 1 | 1 | 0.00 | 3.0598e-01 |
| 4 | 35 | 4 | 1 | 1 | 0.00 | 2.9198e-01 |
| 4 | 35 | 4 | 4 | 1 | 0.88 | 3.4575e-05 |
| 8 | 35 | 4 | 4 | 1 | 1.76 | 1.7323e-05 |
| 16 | 35 | 4 | 4 | 1 | 3.68 | 8.2669e-06 |
| ... | ... | ... | ... | ... | ... | ... |

根据表中结果，我们可发现，当其他条件不改变，使 $P > Q$ 时，执行时间会变长，结果会差一些； $P * Q$ 在 4 以下时 Gflops 较高；N 和 NB 在较大时 Gflops 较高，P 较小时 Gflops 较高； $P = 2^n$ ，即 P 取值为 2 的幂结果会更好。

2. 将组内成员的电脑构建为小“集群”，测试该“集群”的性能:

在进行测试时，两台计算机 CPU 和内存资源都会被一定量的占用，和两台计算机单独运行的表现差异不大，但是需要占用相对较多的网络资源。在同一个局域网下，其他设备的网络速度会降低很多。下图分别为运行情况下，node1 和 node2 资源使用情况。

| 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 | 12000 | 128 | 1 | 1 | 97.48 | 1.1820e+01 |
| 1 | 12000 | 192 | 1 | 1 | 96.50 | 1.1939e+01 |

| | | | | | | |
|-------|-------|-----|---|----|--------|------------|
| 1 | 12000 | 256 | 1 | 1 | 95.67 | 1.2044e+01 |
| 1 | 35 | 4 | 1 | 1 | 0.00 | 1.9016e-01 |
| 2 | 35 | 4 | 1 | 2 | 0.07 | 4.2133e-04 |
| 4 | 35 | 4 | 1 | 4 | 0.17 | 1.8121e-04 |
| 8 | 35 | 4 | 1 | 8 | 0.34 | 9.0008e-05 |
| 16 | 35 | 4 | 1 | 16 | 0.52 | 5.8927e-05 |
| 8 | 35 | 4 | 1 | 4 | 0.24 | 1.2473e-04 |
| 8 | 35 | 4 | 4 | 1 | 1.92 | 1.5804e-05 |
| 8 | 35 | 4 | 8 | 1 | 2.38 | 1.2775e-05 |
| 1 | 14142 | 128 | 1 | 1 | 156.71 | 1.2034e+01 |
| 1 | 14142 | 192 | 1 | 1 | 160.03 | 1.1784e+01 |
| 单机运行 | | | | | | |
| Node1 | | | | | | |
| 1 | 14142 | 192 | 1 | 1 | 149.95 | 1.2577e+01 |
| 1 | 12000 | 192 | 1 | 1 | 91.74 | 1.2560e+01 |
| Node2 | | | | | | |
| 1 | 14142 | 192 | 1 | 1 | 237.08 | 7.9546e+00 |
| 1 | 12000 | 192 | 1 | 1 | 145.89 | 7.8976e+00 |

实验发现，效果不如 node1 进行单机运行，初步考虑可能是由于网络的问题导致运行速度不增反减。

实验数据处理：

1. 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。

调整相关参数

1) 修改 N:

通过学习得知，N 矩阵的规模越大，有效计算所占的比例也越大，系统浮点处理性能越高；占用系统总内存的 80% 左右为最佳。使用公式 $N*N = \text{系统内存} * 80\%$

查看系统内存（以 byte）：

```
free -b
```

2) 修改 NB:

| Processor | NB |
|--|-----|
| Intel® Xeon® Processor X56*/E56*/E7*/E7*/X7* (codenamed Nehalem or Westmere) | 256 |
| Intel Xeon Processor E26*/E26* v2 (codenamed Sandy Bridge or Ivy Bridge) | 256 |
| Intel Xeon Processor E26* v3/E26* v4 (codenamed Haswell or Broadwell) | 192 |
| Intel® Core™ i3/i5/i7-6* Processor (codenamed Skylake Client) | 192 |
| Intel® Xeon Phi™ Processor 72* (codenamed Knights Landing) | 336 |
| Intel Xeon Processor supporting Intel® Advanced Vector Extensions 512 (Intel® AVX-512) instructions (codenamed Skylake Server) | 384 |

图 8 NB 官网参考数据

NB 不可能太大或太小，一般在 256 以下；NB \times 8 一定是 Cache line 的倍数，可以看到 cache line 是 64，一般取 128、192、256

查看 cache line

```
cat coherency_line_size
```

3) 修改 P、Q:

$P \times Q = \text{系统 CPU 数} = \text{进程数}$

根据资料得知， $P \times Q$ 在 4 以下时 Gflops 较高，进程数取本机系统 cpu 数

查看本机 cpu 型号和个数以及 cache line 大小可以使用:

查看型号和个数

```
grep name /proc/cpuinfo
```

查看个数

```
grep 'physical id' /proc/cpuinfo
```

王丹琳:

```
Mem:      total        used        free        shared    buff/cache   available
Swap:      1022357504      88080384      934277120
```

本机内存约为 $2e+9$ ，根据 $N \times N \times 8 = \text{系统总内存} \times 80\%$ 计算，取 $N=14142$

查看逻辑本机 cpu 型号与个数， $P \times Q$ 取 1

```
yyc@yyc-VirtualBox:/program_file/hpl-2.3/bin/name_str$ grep name /proc/cpuinfo
model name      : Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
```

NB 取 128 192 256

```
root@yyyc-VirtualBox:~# cat /sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size
64
```

| 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 | 14142 | 128 | 1 | 1 | 91.43 | 1.2602e+01 |
| 1 | 14142 | 192 | 1 | 1 | 91.16 | 1.2639e+01 |
| 1 | 14142 | 256 | 1 | 1 | 91.58 | 1.2581e+01 |

【这里你们还要不要做啊？】

对调整相关参数性能变高的原因的分析：

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。但是，矩阵规模 N 的增加达到一定阶段后性能却转而下降，这是因为矩阵规模 N 的增加会导致内存消耗量的增加，系统实际内存空间不足，使用缓存，性能会大幅度降低。所以，要尽量增大矩阵规模 N 的同时，又要保证不使用系统缓存。由于，操作系统本身需要占用一定的内存，除了矩阵 A ($N \times N$) 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，所以矩阵 A 占用系统总内存的 80% 左右为最佳，即 $N \times N \times 8 = \text{系统总内存} \times 80\%$ 。我们通过这个式子得到了 N 的优化取值。

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，这里就有一个参数：NB。NB 的选择和软硬件许多因数密切相关，通过查阅资料，NB 不可能太大或太小，数据分块如果过大，则容易造成负载不平衡；但是如果数据分块过小，则通信开销就会很大，同样会影响计算的整体性能。综合以上考虑，要选取一个比较均衡的数值。一般在 256 以下，并且 $NB \times 8$ 一定是 Cache line 的倍数，同时我们也查阅官网，对官网提供的数据进行了参考。我们通过 `cat coherency_line_size` 命令来查看 Cache line 的大小，从而确定 NB 的合理取值。

对于 P, Q: 一个进程对应一个 CPU 可以得到最佳性能，所以 $P \times Q$ 选取系统 CPU 数时获得最好结果，即 $P \times Q = \text{系统 CPU 数} = \text{进程数}$ 。同时 $P \leq Q$ ，即 P 的值尽量取得小一点，列向通信量大于横向通信，结果会更优。

更换 BLAS

BLAS(basic linear algebra subroutine) 是基本线性代数子程序，是目前应用广泛的线性代数核心数学库。BLAS 最初是由 Fortran 语言编写，后来也出现了 C 语言版本的 cBLAS，其函数接口都大致相同，由于当前不同体系芯片厂商生成的处理器差异较大，在后序的发展中出现了两种 BLAS 设计方案：通用型和专用型。专用型 BLAS 一般由特定的芯片公司开发，针对自己平台的芯片特性进行代码优化，在自己平台上的性能往往相对较好，如 Intel 开发的 MKL，AMD 开发的 ACML，NVIDIA 开发的 cuBLAS；通用型 BLAS 一般由非营利性组织开发，在不同平台上都对开源代码 BLAS 进行适当的优化，目前主流的通用型 BLAS

是 ATLAS, GotoBLAS, BLIS 和 OpenBLAS。

所以, 我们想通过换库来尝试实现性能的优化, 我们使用基于 GotoBLAS 优化的 OpenBLAS 来处理矩阵之间的运算。

以下为安装、配置的具体步骤:

下载安装包



使用如下命令解压编译

```
tar zxvf OpenBLAS-0.3.13.tar.gz  
make PREFIX=/home/yyc/ustb/openblas install
```

执行命令, 修改

```
vim Make.name_str
```

按照如下进行修改

```
SHELL      = /bin/sh  
CD          = cd  
CP          = cp  
LN_S        = ln -s  
MKDIR       = mkdir  
RM          = /bin/rm -f  
TOUCH       = touch  
ARCH        = name_str  
TOPdir      = /program_file/ hpl-2.3  
INCdir      = $(TOPdir)/include  
BINdir      = $(TOPdir)/bin/$(ARCH)  
LIBdir      = $(TOPdir)/lib/$(ARCH)  
HPLlib      = $(LIBdir)/libhpl.a  
MPdir       = /program_file/ mpich-3.3.2  
MPinc       = -I$(MPdir)/include
```

```

MPlib      = $(MPdir)/lib/libmpich.so
LAdir      = /program_file/ OpenBLAS
LAinc      = -I$(LAdir)/include
LAlib      = $(LAdir)/libopenblas.a
F2CDEFS    =
HPL_INCLUDES = -I$(INCdir) -I$(INCdir)/$(ARCH) $(LAinc) $(MPinc) -lpthread
HPL_LIBS    = $(HPLlib) $(LAlib) $(MPlib) -lpthread
HPL_OPTS    = -DHPL_CALL_CBLAS
HPL_DEFS    = $(F2CDEFS) $(HPL_OPTS) $(HPL_INCLUDES)
CC          = $(MPdir)/bin/mpicc -lpthread
CCNOOPT     = $(HPL_DEFS)
CCFLAGS     = $(HPL_DEFS) -fomit-frame-pointer -O3 -funroll-loops
LINKER      = $(MPdir)/bin/mpif77
LINKFLAGS   = $(CCFLAGS)
ARCHIVER    = ar
ARFLAGS     = r
RANLIB      = echo

```

修改 HPL_dat 的 PFACTS 与 RFACTS，尝试不同的组合，寻找 HPL 测试所得的实际峰值速度最优的组合：

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

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

测试 NBMINs 为 2 或者 4 的情况，选择最优的 NBMINs：

2 NBMINs

最后，执行命令

sudo ./xhpl

查看优化结果

| 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 | 14142 | 192 | 1 | 1 | 43.09 | 2.6743e+01 |

对比之前的结果，可以发现 HPL 测试所得的实际峰值速度得到了大幅度的提升，所以

更换 BLAS 也可以实现 HPL 的优化。

实验结果与分析：

对于集群性能变差原因的分析：

可能是因为网络的问题导致运行速度不增反减，或者是每个节点的性能不同，集群的性能取决于最差的那个节点。

对调整相关参数性能变高的原因的分析：

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。但是，矩阵规模 N 的增加达到一定阶段后性能却转而下降，这是因为矩阵规模 N 的增加会导致内存消耗量的增加，系统实际内存空间不足，使用缓存，性能会大幅度降低。所以，要尽量增大矩阵规模 N 的同时，又要保证不使用系统缓存。由于，操作系统本身需要占用一定的内存，除了矩阵 $A(N \times N)$ 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，所以矩阵 A 占用系统总内存的 80% 左右为最佳，即 $N \times N \times 8 = \text{系统总内存} \times 80\%$ 。我们通过这个式子得到了 N 的优化取值。

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，这里就有一个参数：NB。NB 的选择和软硬件许多因数密切相关，通过查阅资料，NB 不可能太大或太小，数据分块如果过大，则容易造成负载不平衡；但是如果数据分块过小，则通信开销就会很大，同样会影响计算的整体性能。综合以上考虑，要选取一个比较均衡的数值。一般在 256 以下，并且 $NB \times 8$ 一定是 Cache line 的倍数，同时我们也查阅官网，对官网提供的数据进行了参考。我们通过 `cat coherency_line_size` 命令来查看 Cache line 的大小，从而确定 NB 的合理取值。

对于 P, Q ：一个进程对应一个 CPU 可以得到最佳性能，所以 $P \times Q$ 选取系统 CPU 数时获得最好结果，即 $P \times Q = \text{系统 CPU 数} = \text{进程数}$ 。同时 $P \leq Q$ ，即 P 的值尽量取得小一点，列向通信量大于横向通信，结果会更优

对调整相关参数性能变高的原因的分析：

使用基于 GotoBLAS 优化的 OpenBLAS 来代替 GotoBLAS 处理矩阵之间的运算，性能肯定对变高。

教师评审

| 教师评语 | 实验成绩 |
|-----------------------|------|
| <p>签名:</p> <p>日期:</p> | |