

# 《计算机系统设计》课程设计报告



湖南大学  
HUNAN UNIVERSITY

选题名称: Linpack 标准测试程序及其分析

姓 名: 王倩

学 号: 201708010630

专业班级: 物联 1702

## 概要

LINPACK 是当前评测计算机浮点性能的基准测试程序。Linpack 根据矩阵规模可以分为  $100 \times 100$ ,  $1000 \times 1000$  和  $N \times N$  三种。

HPL: 它提高求解一个稠密线性方程组来测试计算机 Linpack 性能。

计算机硬件的不断的升级带来了系统处理性能持续的提高, 如何的对平台的性能作出正确的判断就是性能测试的主要目标, 业界也有多种测试基准, 有的是基于实际的应用种类如 TPC-C, 有的是测试系统的某一部分的性能, 如测试硬盘吞吐能力的 IOmeter, 测试内存带宽的 stream。

## HPL 理论基础

HPL 通过求解一个稠密线性方程组来测试计算机 LINPACK 性能。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(矩阵大小)、使用 CPU 数目、使用各种优化方法等等来执行该测试程序, 以获取最佳的性能。HPL 采用高斯消元法求解线性方程组。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。HPL 测试结果是 TOP500 排名的重要依据。

## LINPACK 测试

HPL 执行的时间比较长, 特别是当矩阵规模比较大的时候, 而且 LINPACK 性能受各种软硬件以及 HPL 执行参数的影响很大, 所以想取得一个较佳的结果, 需要一套合理的测试方法, 不能盲目进行。

## 影响 LINPACK 性能的因素

LINPACK 性能主要受三个因素的影响, 分别是硬件因素, 软件因素和 HPL 执行参数。

### (1) 硬件因素

硬件因素主要包括 cache 大小和存储系统结构, 访存速度, 处理器性能, 计算机系统的结构以及互连网络的性能等, 这些因素都会影响机器的 LINPACK 性能。

### (2) 软件因素

软件因素主要指的是 MPI 和 BLAS 对 HPL 性能的影响。MPI 常用的有 LAMMPI, MPICH, 和 OpenMPI, 这三种 MPI 性能不一样, 有些针对一些特殊的结构(如: SMP)会进行优化。

编译器的选择也有很大的关系。选择哪种 BLAS 不仅要参考计算机硬件类型, 还要通过实验分析。

### (3) HPL 执行参数

HPL 执行参数很多, 在文件 “HPL.dat” 中进行设置, 其中对 LINPACK 性能影响比较大的是  $P \times Q$ ,  $N$  和  $NB$  ( $P \times Q$  是处理器网络的排列形式,  $N$  是矩阵规模,  $NB$  是矩阵分块的大小), 测试时需要不断的进行调整。

由于硬件一般比固定, 所以我们在 LINPACK 测试时主要调整的是软件以及 HPL 执行参数。

## LINPACK 测试方法

(1) 根据各参数对 LINPACK 性能影响的大小, 将其分为 A 类参数和 B 类参数。A 类参数包括  $P \times Q$ ,  $N$  和  $NB$ , 其他都是 B 类参数。A 类参数对 LINPACK 性能影响很大, B 类参数影响较小, 所以测试的时候先找到最佳的  $P \times Q$ , 再确定最佳的  $N$ , 然后是  $NB$ , 当三个 A 类参数定下来之后, 再确定 B 类参数。

(2) 根据定义为矩阵运算时间与其运算量之比的效率因子很大程度上相关与矩阵分块大小, 而与矩阵规模本身关系微乎其微这一规律, 通过扫码小规模运算矩阵效率, 来确定大规模并行测试中分块大小  $NB$ , 已达到缩短测试时间的目的。

上述两种方法的缺点:

它们都只把 LINPACK 的浮点速率作为唯一的评判标准, 而忽略了很多细节, 比如通信时间的比重, 矩阵运算时间的比重等等。

改进的测试方法:

基于计时系统的测试方法, 对原有的 HPL 计时系统进行改进, 提取更加丰富的关键时间参数。通过计时系统, 可以快速定位最佳测试平台软硬件配置和 HPL 执行参数, 以达到快速完成 LINPACK 测试的目的。

## 基于技术系统的 LINPACK 测试

测试的一般步骤:

(1) 确定互联网络, 在不同的互联网络中运用同样的 HPL 程序。通过比较时间参数, 选择哪种互联网络。

(2) 确定 BLAS, 并且提取各个 BLAS 对应性能较好的  $NB$  值。在单处理器,  $N$  值较小,  $NB$  取值范围较宽的情况下进行测试和比较, 确定最佳的 BLAS 函数库, 时间参数反应出 BLAS 的优劣。

(3) 确定 MPL。  $P \times Q$  取一个适中的值, 通过提取时间参数, 确定 MPI 库。

(4) 确定  $P \times Q$ 。

(5) 确定  $N$ ,  $NB$ 。

(6) 确定广播算法, DEPTH 以及其他 HPL 的执行参数。

## 测试过程

### 硬件准备

准备硬件平台的配置

升级到最新的 BIOS、BMC 等版本

调整对性能有影响的参数设置

### 软件准备

操作系统的准备。由于 OS 自身也会占用系统资源, 因此一般会使用 linux 作为 linpack 测试的 OS 平台, 采用最新的内核版本的 linux, 这样可以充分的发挥出硬件的新的特性, 发挥出平台的计算性能; 在系统启动后, 将一些没有必要的系统守护进程去掉, 可以运行 `ntsysv` 命令, 关闭除了 `irqbalance` 和

messagebus. 之外的系统服务进程, 也可以节省系统的资源; 并且将操作系统启动到第 3 级, 不要进入图形方式。3、linpack 的参数设置

### 运算阶数 N 的选择

由于在计算的过程中, 会分配、占有一定的内存空间, 因此依据内存容量合适的设置 N 的数值, 会得到较为准确的计算性能数据。如果 N 设置较小, 内存不能充分利用, 则处理能力不能发挥; 如果 N 设置较大, 内存空间不能满足需求, 则需要经常的执行硬盘读写, 从而会有处理器的等待时间, 计算时间会延长, 测试得到计算性能结果也会受到影响。根据内存容量大小, 对应的 N 的数值有如下的参考关系。

例如在内存容量为 4G 时, 设置 N 为 22000 较为合理, 这样内存分配较为合理, 因此在 22000 阶时, 可能得到最大计算性能值。

### 配置文件的编写

下面是 linpack 运行的参数配置文件的例子, 其中包括一个参数。

- 计算的点数, 原则上是计算的点数越多, 则会遍历多种计算的性能情况, 更能找到最好的性能点, 但是点数越多则运算时间越长。
- 点数的分布。即设定几个不同的阶数值, 一般是在 N 附近的时候的阶数分布较为密, 以便找到最佳性能数据。
- 每个计算点的计算次数, 为了减少测试误差, 增加每点的计算次数取其平均值, 得到比较可信的性能数据。
- 设置内存的对齐尺寸, 内存分配的时候的内存对其方式, 可以提高内存的读取的效率, 提高性能测试结果, 但是设置过大会产生一定的内存空间的浪费, 一般为 4KB 或 8KB

### 运行

Intel 提供了基于 IA 架构平台优化后的可执行版本 3.0.1, 不需要下载源代码再通过编译器, 编译优化, 因此使用起来比较的简单和方便。现在可以执行的平台有

xlinpack\_itanium 在 64 位安腾 2 平台上的可执行程序

xlinpack\_xeon32 在 32 位至强 DP 和至强 MP 平台上可执行程序, 同时可支持有或没有 SSE3 指令集的情况。

xlinpack\_xeon64 可运行在 64 位扩展技术的至强平台上

xlinpack\_mc64 运行在 Intel Core 2 Duo 和 Woodcrest 新一代处理器上的程序。

可以编写一个 shell 的脚本文件, 这样可以灵活的控制运行的过程, 运行结果的记录等。例如下面的脚本文件的例子, 先设置系统内的对称多处理器的数目, 在使用参数设置文件的参数运行 linpack, 并把结果输出到一个文本文件内。

```
#export OMP_NUM_THREADS=2 echo "This is a SAMPLE run script.
Change it to reflect the correct number"echo "of CPUs/threads, proble
m input files, etc.." datedate > lin_xeon32.txt./xlinpack_xeon32 lininput
_xeon32 >> lin_xeon32.txtdate >> lin_xeon32.txtecho -n "Done: "date
```

## 结果查看

根据上面的运行脚本文件，运行结果输出到文件 lin\_xeon32.txt 内。

## 修改 LINPACK 源程序是遇到的问题

Make 的 command 是：make arch=test 这里的 arch=test 实际上是给变量 arch 赋值为 test，会作用到 Makefile 中，因为 Makefile 文件中有用到这个 arch 变量。

```
install      : startup refresh build
#
startup      :
    $(MAKE) -f Make.top startup_dir    arch=$(arch)
    $(MAKE) -f Make.top startup_src    arch=$(arch)
    $(MAKE) -f Make.top startup_tst    arch=$(arch)
    $(MAKE) -f Make.top refresh_src    arch=$(arch)
    $(MAKE) -f Make.top refresh_tst    arch=$(arch)
```

这里的 install 依赖 startup refresh build 三项。而 startup 由之后的命令生成。Make -f 命令后指定了执行的是 Make.top 文件中的 Startup\_dir 项，并且参数项

```
arch=$(arch)
```

```
startup_dir
```

将 arch 的值传递到 Make.top 中的 arch 中那么我们现在转到 Make.top 中的项，如下：

```
startup_dir :
- $(MKDIR) include/$(arch)
- $(MKDIR) lib
- $(MKDIR) lib/$(arch)
- $(MKDIR) bin
- $(MKDIR) bin/$(arch)
```

意为在创建相关的文件夹路径，包括在 include 下创建 test 目录，创建 lib 目录并在 lib 目录下创建 test 目录，创建 bin 目录，并在 bin 目录下创建 test 目录。

```
startup_src :
- $(MAKE) -f Make.top leaf le=src/auxil      arch=$(arch)
- $(MAKE) -f Make.top leaf le=src/blas       arch=$(arch)
- $(MAKE) -f Make.top leaf le=src/comm       arch=$(arch)
- $(MAKE) -f Make.top leaf le=src/grid       arch=$(arch)
- $(MAKE) -f Make.top leaf le=src/panel      arch=$(arch)
- $(MAKE) -f Make.top leaf le=src/pauxil     arch=$(arch)
- $(MAKE) -f Make.top leaf le=src/pfact     arch=$(arch)
- $(MAKE) -f Make.top leaf le=src/pgesv     arch=$(arch)
```

接下来是，刚开始不明白 leaf 是什么意思，以为是 make 相关命令还是什么，楞查也没查到，在 Make.top 文件后面发现

```
leaf :
- ( $(CD) $(le) ; $(MKDIR) $(arch) )
- ( $(CD) $(le)/$(arch) ; \
    $(LN_S) $(TOPdir)/Make.$(arch) Make.inc )
```

这才恍然大悟，原来这里的 leaf 类似于 clean 的伪目标项，比如 make clean 就是执行 clean 操作，清除相关的文件，而 make - leaf 则就是执行 leaf 下定义的相关命令操作。

```
le=src/auxil      arch=$(arch)
```

后面的都不过是对有关变量的赋值

ln 是做链接的命令  
ln -s 源文件 目标文件 做软链接  
和快捷方式只能共享执行文件不同，linux通过[链接文件](#)能共享几乎所有类型的文件。

语句而

已。转入文件路径下，在该路径下创建 test 文件夹。

转入到 src/auxil/test/路径下

LN\_S 在 Make.test 中被赋值为 ln -s 该命令为为某个文件创建链接。

所以即将 hpl/Make.test 文件创建一个链接文件到当前路径，即 src/auxil/test/Make.inc 后面的分析与以上类似。

```
refresh_src :
- $(CP) makes/Make.auxil      src/auxil/$(arch)/Makefile
- $(CP) makes/Make.blas      src/blas/$(arch)/Makefile
- $(CP) makes/Make.comm      src/comm/$(arch)/Makefile
- $(CP) makes/Make.grid      src/grid/$(arch)/Makefile
- $(CP) makes/Make.panel     src/panel/$(arch)/Makefile
- $(CP) makes/Make.pauxil    src/pauxil/$(arch)/Makefile
- $(CP) makes/Make.pfact     src/pfact/$(arch)/Makefile
- $(CP) makes/Make.pgesv     src/pgesv/$(arch)/Makefile
```

```
refresh_tst :
- $(CP) makes/Make.matgen    testing/matgen/$(arch)/Makefile
- $(CP) makes/Make.timer     testing/timer/$(arch)/Makefile
- $(CP) makes/Make.pmatgen   testing/pmatgen/$(arch)/Makefile
- $(CP) makes/Make.ptimer    testing/ptimer/$(arch)/Makefile
- $(CP) makes/Make.ptest     testing/ptest/$(arch)/Makefile
```

这两项 refresh 执行的是文件的拷贝工作。可以看出，作者将功能性的代码和工程性的代码分隔开。又是结构化，层次化，模块化的思想体现的淋漓尽致，软工真的是 very nice。

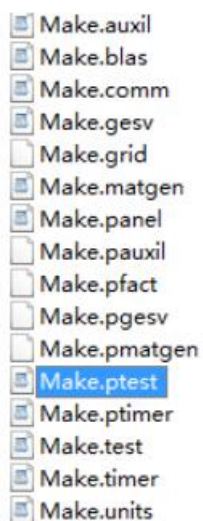
而我们通常在代码开发完成后的优化工作中才做这些工作，最后才发现有时候代码写的关联性耦合性极强，像一团浆糊，已经相当无奈了。

#### 4. Build 的过程

```
build_src :
( $(CD) src/auxil/$(arch);      $(MAKE) )
( $(CD) src/blas/$(arch);      $(MAKE) )
( $(CD) src/comm/$(arch);      $(MAKE) )
( $(CD) src/grid/$(arch);      $(MAKE) )
( $(CD) src/panel/$(arch);      $(MAKE) )
( $(CD) src/pauxil/$(arch);     $(MAKE) )
( $(CD) src/pfact/$(arch);     $(MAKE) )
( $(CD) src/pgesv/$(arch);     $(MAKE) )

#
build_tst :
( $(CD) testing/matgen/$(arch); $(MAKE) )
( $(CD) testing/timer/$(arch);  $(MAKE) )
( $(CD) testing/pmatgen/$(arch); $(MAKE) )
( $(CD) testing/ptimer/$(arch); $(MAKE) )
( $(CD) testing/ptest/$(arch);  $(MAKE) )
```

转入相应的文件夹路径，执行各自的 make，生成相应的目标文件，最终是要生成一个库文件 libhpl.a 如何实现的呢？找啊找啊，没一下看出来，于是查看了 makes 文件下的每个 makefile 文件，在





```

xhpl          = $(BINDir)/xhpl
#
## Object files #####
#
HPL_pteobj    = \
    HPL_pddriver.o      HPL_pdinfo.o      HPL_pdtest.o
#
## Targets #####
#
all          : dexe
#
dexe         : dexe.grd
#
$(BINDir)/HPL.dat : ../HPL.dat
    ( $(CP) ../HPL.dat $(BINDir) )
#
dexe.grd: $(HPL_pteobj) $(HPLlib)
    $(LINKER) $(LINKFLAGS) -o $(xhpl) $(HPL_pteobj) $(HPL_LIBS)
    $(MAKE) $(BINDir)/HPL.dat
    $(TOUCH) dexe.grd
..

```

中有发现：

我们知道最后生成的可执行文件的名字为 xhpl，现在这个字样终于出现了！，有没有好激动！找了 so long time，我只觉得比较惨， dumass！ 现在看它是怎么生成的， 根据依赖项，看这

```

$(LINKER) $(LINKFLAGS) -o $(xhpl) $(HPL_pteobj) $(HPL_LIBS)

```

此处的变量 LINKER, LINKFLAGS 都能在 Make.test 中找到。

```

$(HPL_pteobj) $(HPL_LIBS)

```

这条指令执行的是链接，将上面两个链接在一起，生成为最终的输出文件，\$HPL\_pteobj 在本 makefile 中定义的，即为：

```

HPL_pteobj    = \
    HPL_pddriver.o      HPL_pdinfo.o      HPL_pdtest.o

```

那么后面的\$HPL\_LIBS 是什么呢？ 它在 Make.test 文件中被赋值过。

```

HPL_LIBS      = $(HPLlib) $(LAlib) $(MPLib)

```

```

HPLlib        = $(LIBdir)/libhpl.a

```



我们可以看到，它由三项组成，

HPLlib 其实是 hpl 的库文件

```
LAlib = $(LAdir)/libgoto2.a $(LAdir)/libgoto2.so
```

LAlib 其实是 GotoBLAS 的库文件

```
MPlib = $(MPdir)/lib/libmpi.so
```

MPlib 其实是 MPI 的库文件，这里用的是 openmpi

GotoBLAS 与 MPI 的库都是在各自编译的时候生成的，那么 HPL 的库在编译的时候怎么生成的呢？目前我们并没有看到其生成过程。

但我们此处已经知道了 xhpl 是如何生成的了，稍微值得欣慰一点。那么接下来寻找传说中的 libhpl.a 是如何生成的？

查看 makes 文件夹下的其他 makefile 文件，比如这个 Make.blas 文件会发现，有这么一段：

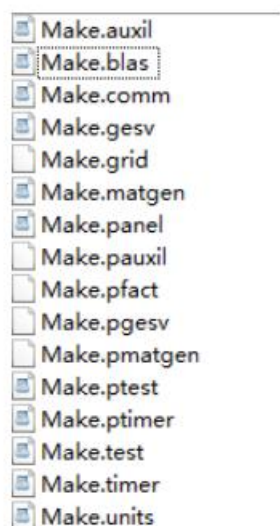
```
HPL_blaobj = \
    HPL_dcopy.o      HPL_daxpy.o      HPL_dscal.o
    \
    HPL_idamax.o     HPL_dgemv.o      HPL_dtrsv.o
    \
    HPL_dger.o       HPL_dgemm.o      HPL_dtrsm.o
#
## Targets #####
#
all      : lib
#
lib      : lib.grd
#
lib.grd : $(HPL_blaobj)
          $(ARCHIVER) $(ARFLAGS) $(HPLlib) $(HPL_blaobj)
          $(RANLIB) $(HPLlib)
          $(TOUCH) lib.grd
```

变量 HPL\_blaobj 指的是目标文件，如上图所示，变量 [Math Processing Error]ARCHIVERARFLAGS 以及 \$HPLlib 在 Make.test 文件中都有赋值。

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

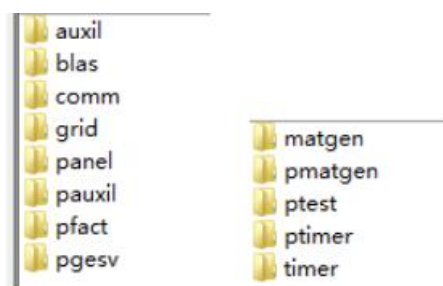
由此可看出，要执行的命令实际上是：`ar r [Math Processing Error](LIBdir)/libhpl.a(HPL_blaobj)`

现在我们知道了 `ar` 是创建库的命令，`r` 表示向库中插入模块，也就是说，会将后面的那些 `HPL_dcopy.o`, `HPL_daxpy.o`, `HPL_dscal.o` 等作为模块插入到 `libhpl.a` 库文件中。



由此可以知道，这些文件中如果也有相似的段落，

也会将生成的相应的 `.o` 文件插入到 `libhpl.a` 库文件中，一步一步，逐渐生成完整地 `libhpl.a` 文件，我们前面看到的代码的模块化，无论是 `src` 下的，还是 `testing` 下的，都是将各自模块的 `.o` 文件在各自文件夹下的 `makefile` 编译的时候，通过 `ar` 命令利用参数选项 `r`，插入到 `libhpl.a` 库中。



前面说过，看到了 `xhpl` 字样，那么顺着它寻找我们的 `main` 函数，先找到对应的那个 `makefile` 文件，`/home/zgz/sourcecode/linpack/hpl/testing/ptest/test` 下的 `Make.ptest` 文件。

```

dexe.grd: $(HPL_nteobj) $(HPLlib)
$(LINKER) $(LINKFLAGS) -o $(xhpl) $(HPL_pteobj) $(HPL_LIBS)
$(MAKE) $(BINDir)/HPL.dat
$(TOUCH) dexe.grd

#
# #####
#
HPL_pddriver.o      : ../HPL_pddriver.c      $(INCdep)
$(CC) -o $@ -c $(CCFLAGS) ../HPL_pddriver.c
HPL_pdinfo.o        : ../HPL_pdinfo.c        $(INCdep)
$(CC) -o $@ -c $(CCFLAGS) ../HPL_pdinfo.c
HPL_pdtest.o        : ../HPL_pdtest.c        $(INCdep)
$(CC) -o $@ -c $(CCFLAGS) ../HPL_pdtest.c

```

可以看到，链接时，除了库文件外，还有一些目标文件，对应的是紫色的部分。

那么 main 函数在哪里呢？打开对应的.c 文件看一下吧。有没有感到连灵魂都在颤抖？

呃。。。其实，它真的就在 HPL\_pddriver.c 中！！！！

另外在 comm 文件夹下，有

Local Name	/	Size	Type
Linux_PII_CBLAS			文件夹
test			文件夹
HPL_1ring.c		7,404	C Source
HPL_1rinM.c		7,784	C Source
HPL_2ring.c		7,760	C Source
HPL_2rinM.c		8,157	C Source
HPL_bcast.c		5,197	C Source
HPL_binit.c		4,635	C Source
HPL_blong.c		12,975	C Source
HPL_blonM.c		15,845	C Source
HPL_bwait.c		4,686	C Source
HPL_copyL.c		4,591	C Source
HPL_packL.c		8,907	C Source
HPL_recv.c		6,043	C Source
HPL_sdrv.c		9,177	C Source
HPL_send.c		5,944	C Source

而在 grid 文件夹路径下，有

Local Name	Size	Type
Linux_PII_CBLAS		文件夹
test		文件夹
HPL_all_reduce.c	5,168	C Source
HPL_barrier.c	4,041	C Source
HPL_broadcast.c	6,140	C Source
HPL_grid_exit.c	4,778	C Source
HPL_grid_info.c	5,280	C Source
HPL_grid_init.c	7,873	C Source
HPL_max.c	5,001	C Source
HPL_min.c	5,001	C Source
HPL_pnum.c	4,700	C Source
HPL_reduce.c	7,480	C Source
HPL_sum.c	4,975	C Source

可以看出，这两处的文件，跟 MPI 通信的定义和实现有关，因此修改程序时，重点关注。

## 总结

提出了一种基于计时系统的 LPNPACK 测试方法，这种测试方法可以提取详细的时间参数，更好的引导 LINPACK 的快速测试。我认为这种方法是非常有效的。