湖南大學

HUNAN UNIVERSITY

《 计算机系统设计 》

题 目	Linpack 标准测试程序及其分析
专 业	智能科学与技术
班 级	智能 1702
姓 名	唐昌均
学 号	201708010712

1 简介

1.1 引言

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。

1.2 始用途

LINPACK 主要的特色是:

a 率先开创了力学 (Mechanics) 分析软件的制作。

b 建立了将来数学软件比较的标准。

c 提供软件链接库， 允许使用者加以修正以便处理特殊问题， (当然程序名称必须改写， 并应注明修改之处， 以尊重原作者， 并避免他人误用。)

d 兼顾了对各计算机系统的通用性， 并提供高效率的运算。

至目前为止， LINPACK 还是广泛地应用于解各种数学和工程问题。 也由于它高效率的运算， 使得其它几种数学软件例如 IMSL、MATLAB 纷纷加以引用来处理矩阵问题， 所以足见其在科学计算上有举足轻重的地位。

1.3 性能测试

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。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。

计算峰值

随着产品硬件的不断的升级，整个的计算能力也以数量级的速度提升。衡量计算机性能的一个重要指标就是计算峰值，例如浮点计算峰值，它是指计算机每秒钟能完成的浮点计算最大次数。包括理论浮点峰值和实测浮点峰值：

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

理论计算公式如下：**理论浮点峰值 = CPU 主频×CPU 每个时钟周期执行浮点运算次数×CPU 数量**

实测浮点峰值是指 Linpack 测试值，也就是说在这台机器上运行 Linpack 测试程序，通过各种调优方法得到的最优的测试结果。实际上在实际程序运行过程中，几乎不可能达到实测浮点峰值，更不用说达到理论浮点峰值了。这两个值只是作为衡量机器性能的一个指标，用来表明机器处理能力的一个标尺和潜能的度量。

CPU 每个时钟周期执行浮点运算的次数是由处理器中浮点运算单元的个数及每个浮点运算单元在每个时钟周期能处理几条浮点运算来决定的，下表是常见 CPU 的每个时钟周期执行浮点运算的次数。

IBM Power6	Pentium	Opteron	PA-RISC	SGI MIPS	Xeon	Itanium
4	1	2	4	2	4	4

2 源码文件分析

Makefile.test 文件中定义了一些后面会用到的变量。

如目标架构名称：

```
#
# - Platform identifier ------
# ------------------------------
#
ARCH         =test
#
TOPdir       = /home/zgz/sourcecode/linpack/hpl
INCdir       = $(TOPdir)/include
BINdir       = $(TOPdir)/bin/$(ARCH)
LIBdir       = $(TOPdir)/lib/$(ARCH)
#
HPLlib       = $(LIBdir)/libhpl.a
```

HPL 相关的各种路径 ：

```
# ----------------------------------------
# - Message Passing library (MPI) --------
# ----------------------------------------
# MPinc tells the  C  compiler where to fi
# header files,  MPlib  is defined  to be
# used. The variable MPdir is only used fo
#
MPdir        = /home/zgz/openmpi
MPinc        = -I$(MPdir)/include
MPlib        = $(MPdir)/lib/libmpi.so
```

MPI 的相关路径：

```
# ----------------------------------------------------------
# - Linear Algebra library (BLAS or VSIPL) -----------------
# ----------------------------------------------------------
# LAinc tells the  C  compiler where to find the Linear Alge
# header files,  LAlib  is defined  to be the name of  the l
# used. The variable LAdir is only used for defining LAinc a
#
LAdir        = /home/zgz/sourcecode/linpack/GotoBLAS2
LAinc        =
LAlib        = $(LAdir)/libgoto2.a $(LAdir)/libgoto2.so
```

数学库 BLAS 的路径：

```
#
# - Compilers / linkers - Optimization flags --------
# -----------------------------------------------------
#
CC           = /home/zgz/openmpi/bin/mpicc
```

编译器的路径：

```
#
LINKER       = /home/zgz/openmpi/bin/mpif77
```

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)
```

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。

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

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。

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以类型的文件。

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。

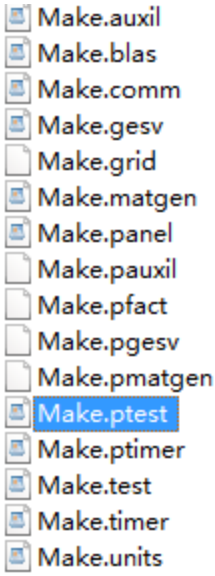
```
refresh_src      :
        - $(CP) makes/Make.auxil    src/auxil/$(arch)/Makefile
        - $(CP) makes/Make.blas     src/blas/$(arch)/Makefile
        - $(CP) makes/Make.comm     src/comm/$(arch)/Makefile
        - $(CP) makes/Make.grid     src/grid/$(arch)/Makefile
        - $(CP) makes/Make.panel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

以上两项 refresh 执行的是文件的拷贝工作。可以看出，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。

以下是 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 Make.ptest 中我们发现：




```
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
```

此处的变量 LINKER，LINKFLAGS 都能在 Make.test 中找到。这条指令执行的是链接，将

```
$(HPL_pteobj) $(HPL_LIBS)
```

这两个链接在一起，生成

```
xhpl                = $(BINdir)/xhpl
```

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)
```

MPIlib 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 ARCHIVERARCHIVERARFLAGS 以及 $HPLlib 在 Make.test 文件中都有赋值。

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

可以通过以上文件看到，Linpack 测试程序不是一个简单的程序，他包含了很多相关的文件，运行环境，连接程序，只有通过这些复杂的操作和精密的逻辑，最后才能对一台高性能计算机做出相对客观的评价。