# FFTW

[dwt@node0 fftw-3.3.2]$ ./configure --prefix=/home/dwt/software/fftw --enable-mpi LDFLAGS=-L/home/dwt/software/mpich/lib/ CPPFLAGS=-I/home/dwt/software/mpich/include/

[dwt@node0 fftw-3.3.2]$ make

[dwt@node0 fftw-3.3.2]$ make install

测试mpi的fftw：test.cpp

#include <fftw3-mpi.h>

int main(int argc, char \*\*argv)

{

const ptrdiff\_t N0 = 10, N1 = 10;

fftw\_plan plan;

fftw\_complex \*data; //local data of course

ptrdiff\_t alloc\_local, local\_n0, local\_0\_start, i, j;

int myid;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myid);

fftw\_mpi\_init();

printf("2 node % come to this place.\n", myid);

/\* get local data size and allocate \*/

alloc\_local = fftw\_mpi\_local\_size\_2d(N0, N1, MPI\_COMM\_WORLD,

&local\_n0, &local\_0\_start);

data = (fftw\_complex \*) fftw\_malloc(sizeof(fftw\_complex) \* alloc\_local);

printf("%i\n", local\_n0);

/\* create plan for forward DFT \*/

plan = fftw\_mpi\_plan\_dft\_2d(N0, N1, data, data, MPI\_COMM\_WORLD,

FFTW\_FORWARD, FFTW\_ESTIMATE);

/\* initialize data to some function my\_function(x,y) \*/

for (i = 0; i < local\_n0; ++i) for (j = 0; j < N1; ++j){

data[i\*N1 + j][0]=local\_0\_start;

data[i\*N1 + j][1]=i;

}

/\* compute transforms, in-place, as many times as desired \*/

fftw\_execute(plan);

fftw\_destroy\_plan(plan);

fftw\_free(data);

MPI\_Finalize();

printf("finalize\n");

return 0;

}

[dwt@node0 tests]$ mpicxx -o test test.cpp -I/home/dwt/software/fftw/include -lfftw3\_mpi -lfftw3 -lm -L/home/dwt/software/fftw/lib

[dwt@node0 tests]$ mpiexec –machinefile hosts –n 3 ./test

**问题1：如果用mpicc编译，则报错undefined reference to `\_\_gxx\_personality\_v0。**

报任何MPI的问题都是由于mpi库没安装成功。

# BLAS

CBLAS是BLAS的C语言接口。BLAS的全称是Basic Linear Algebra Subprograms，中文大概可以叫做基础线性代数子程序。主要是用于向量和矩阵计算的高性能数学库。本身BLAS是用Fortran写的，为了方便C/C++程序的使用，就有了BLAS的C接口库CBLAS。

CBLAS/BLAS分为3个level，level1是用于向量的计算，level2是用于向量和矩阵之间的计算，level3是矩阵之间的计算，比如计算矩阵的乘法就是属于level3。

常用的BLAS库有GOTO、Atlas、ACML、ESSL、MKL等，公认GOTO库性能最优。

安装BLAS（Fortran版）和CBLAS（标准C），<http://www.netlib.org/blas/>

下载并解压tar zxvf blas.tgz和cblas.tgz

1. 编译BLAS

根据系统修改make.inc和Makefile(Linux不用修改)，make，生成blas\_LINUX.a。

1. 编译CBLAS

ln –s Makefile.ARCH Makefile.in ARCH为相应的系统(Linux不用修改)

修改BLLIB为blas\_LINUX.a的路径，CBLIB为生成cblas\_LINUX.a的路径。

BLLIB = /home/dwt/software/BLAS/blas\_LINUX.a

CBLIB = ../lib/cblas\_$(PLAT).a

make，生成lib/cblas\_LINUX.a。

1. 测试CBLAS：

[dwt@node0 CBLAS]$ cp include/cblas.h src

[dwt@node0 CBLAS]$ cp include/cblas\_f77.h src

[dwt@node0 CBLAS]$ cd examples

[dwt@node0 examples]$ make

[dwt@node0 examples]$ ./cblas\_ex1

# HPC challenge benchmark

主要Benchmark软件包包括： LINPACK，HPC Challenge Benchmark （HPCC）、 NAS 并行基准测试（NPB）、SPEC HPC测试，IDC平衡评价指标（Balanced Rating）、有效系统性能测试（ESP）、集成的计算机性能分析（IPACS）、PerfExplorer。

## 安装

安装hpcc前，必须先安装好mpich和blas（或cblas）。具体安装步骤参考README和hpl目录下的README和INSTALL文件。

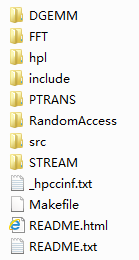
从<http://icl.cs.utk.edu/hpcc/>下载最新稳定版本。

[dwt@node0 software]$ tar zxvf hpcc-1.4.1.tar.gz

[dwt@node0 software]$ cd hpcc-1.4.1

[dwt@node0 hpcc-1.4.1]$ ls

1. **目录结构**

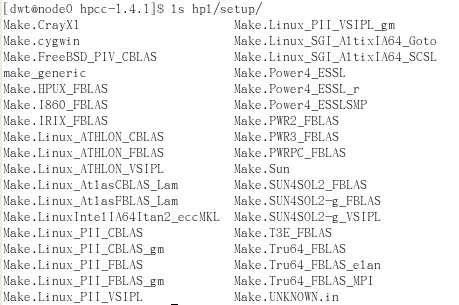


压缩包内含有7个基准程序的目录，头文件和源文件目录，\_hpccinf.txt示例文件，Makefile，和README。首先编译程序，生成可执行文件，然后运行时配置，即修改输入文件内容，最后运行，输出结果。

1. **生成Make.arch**

在hpl目录下新建Make.arch脚本，脚本须以Make.做前缀，后缀arch为目标系统，如Unix系统则命名为Make.Unix。可从hpl/setup目录下找到适合当前环境的脚本，拷贝到hpl目录，修改mpich和blas（cblas）的配置内容和文件名。

[dwt@node0 hpcc-1.4.1]$ ls hpl/setup/

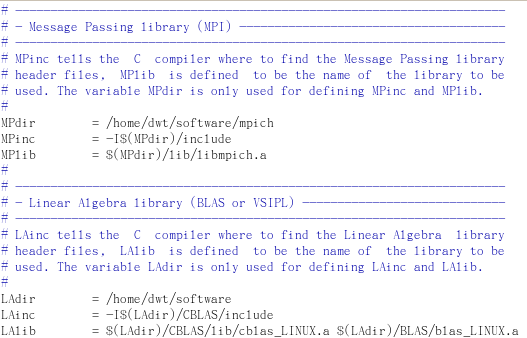


选择Make.Linux\_PII\_CBLAS\_gm代表Linux操作系统、PII平台、采用CBLAS库且MPI为GM。

改名为Make.Linux。

[dwt@node0 hpcc-1.4.1]$ cp hpl/setup/Make.Linux\_PII\_CBLAS\_gm hpl/Make.Linux

[dwt@node0 hpcc-1.4.1]$ vi hpl/Make.Linux



MPdir：MPI安装路径

MPinc：MPI头文件路径

MPlib：libmpich.a库文件路径

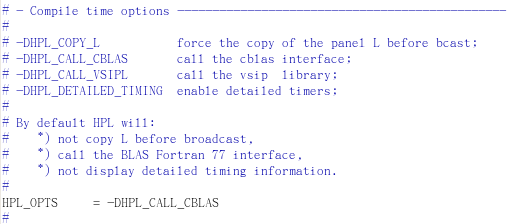
MPICH为$(MPdir)/lib/libmpich.a

OpenMPI为$(MPdir)/lib/libmpi.so

LAdir：CBLAS（或BLAS）安装路径

LAinc：CBLAS头文件路径

LAlib：库文件路径



HPL\_OPTS：包含采用什么库、是否打印详细的时间、是否在L广播之前拷贝L，若采用FLBAS库则置为空，采用CBLAS库为“-DHPL\_CALL\_CBLAS”，采用VSIPL为“-DHPL\_CALL\_VSIPL”

-DHPL\_COPY\_L：广播前将L列先拷贝到一个连续的缓冲区，由于避免了拷贝，有利于HPL生成用户定义的MPI数据格式（对性能影响不大）；

-DHPL\_CALL\_CBLAS：调用CBLAS接口；

-DHPL\_CALL\_VSIPL：调用VSIP库；

-DHPL\_DETAILED\_TIMING：打印每一步所需的时间。

HPL的默认选项为：

1. 广播前不拷贝L；
2. 调用BLAS Fortran 77接口；
3. 不显示编译细节。
4. **编译并运行**

[dwt@node0 hpcc-1.4.1]$ make arch=Linux 2>&1 | tee t.txt

设置arch=Linux，则会寻找hpl/Make.Linux，利用其生成hpcc可执行文件，并在hpl的各目录下生成名为Linux的目录以保存编译后的文件。

可用[dwt@node0 hpl]$ make clean\_arch\_all arch=Linux删除所有生成的目录和文件。

不设置arch=Linux，则会寻找hpl/Make.UNKNOWN。

如果编译成功，则在hpcc-1.4.1目录下生成hpcc可执行文件。

**问题1：make后重新make会报各种错**

[dwt@node0 hpcc-1.4.1]$ make clean arch=Linux

[dwt@node0 hpcc-1.4.1]$ cd hpl;make clean\_arch\_all arch=Linux;cd ..

[dwt@node0 hpcc-1.4.1]$ make arch=Linux 2>&1 | tee t.txt

[dwt@node0 hpcc-1.4.1]$ cp \_hpccinf.txt hpccinf.txt

hpcc的输入文件名须为hpccinf.txt，因此修改示例文件名。

**问题2：直接运行hpcc会报错，提示需要4个进程才能运行。**

[dwt@node0 hpcc-1.4.1]$ ./hpcc

HPL WARNING from process # 0, on line 313 of function HPL\_pdinfo:

>>> cannot open file hpccinf.txt <<<

[dwt@node0 hpcc-1.4.1]$ mpiexec -machinefile hosts -n 4 ./hpcc

HPC Challenge benchmark的运行取决于MPI执行和系统细节。命令行mpirun –np 4 ./hpcc。

* mpirun为MPI执行命令，不同系统可能为aprun，mpiexec，mprun，poe等；
* -np 4指明采用4个MPI进程，进程数必须足够到覆盖所有hpccinf.txt里的进程。
* hpcc为HPC Challenge的可执行文件。

运行完后，生成hpccoutf.txt，其包含基准程序的输出，需上传到HPC Challenge网站。

[dwt@node0 hpcc-1.4.1]$ make clean arch=Linux

1. **编译参数**

可通过C预编译器符号配置和编译HPC Challenge源码，如CCNOOPT和CCFLAGS。CCNOOPT不用优化以保证精确生成系统参数，CCFLAGS用于其他需要优化的文件。

* HPCC\_FFT\_235：如果设置，则FFTE代码使用的矢量大小和处理器个数，不为2的次方，而是2，3，5的次方；
* HPCC\_FFTW\_ESTIMATE：如果设置，定义外部FFTW库的调用方式，采用FFTW\_ESTIMATE，而不是FFTW\_MEASURE方式调用FFTW planning routine，可能导致性能差但处理时间缩短，或内存碎片降低。
* HPL\_USE\_GETPROCESSTIMES：如果设置，调用windows的GetProcessTimes()方法测量CPU经过时间；
* USE\_MULTIPLE\_RECV：如果设置，同时发送多个非阻塞接收。默认只发生一个非阻塞接收；
* RA\_SANDIA\_NOPT：如果设置，则不采用通用RandomAccess的HPC Challenge标准算法。替换算法为Sandia National Laboratory提供。利用MPI进程组成超立方体传输消息。
* RA\_SANDIA\_OPT2：如果设置，，则不采用通用RandomAccess的HPC Challenge标准算法。替换算法为Sandia National Laboratory提供。将处理器个数优化为2的幂次。在发送数据前对数据排序，新的循环前展开数据。如果处理器个数不为2的幂次则与RA\_SANDIA\_NOPT相同。
* USING\_FFTW：如果设置，则不采用标准HPC Challenge FFTE，而是调用FFTW库，光定义USING\_FFTW还不够，还要在FFTW头文件中设置合适的标志。

Makefile

.PHONY : clean

clean :

-rm edit $(objects)

.PHONY意思表示clean是一个“伪目标”，。而在命令前面加了一个小减号的意思就是，也许某些文件出现问题，但不要管，继续做后面的事。当然，clean的规则不要放在文件的开头，不然，这就会变成make的默认目标，相信谁也不愿意这样。不成文的规矩是——“clean从来都是放在文件的最后”。

make命令会在当前目录下按顺序找寻文件名为“GNUmakefile”、“makefile”、“Makefile”的文件，找到了解释这个文件。在这三个文件名中，最好使用“Makefile”这个文件名，因为，这个文件名第一个字符为大写，这样有一种显目的感觉。最好不要用“GNUmakefile”，这个文件是GNU的make识别的。有另外一些make只对全小写的“makefile”文件名敏感，但是基本上来说，大多数的make都支持“makefile”和“Makefile”这两种默认文件名。

当然，你可以使用别的文件名来书写Makefile，比如：“Make.Linux”，“Make.Solaris”，“Make.AIX”等，如果要指定特定的Makefile，你可以使用make的“-f”和“--file”参数，如：make -f Make.Linux或make --file Make.AIX。

objects := $(wildcard \*.o) 让通配符在变量中展开

VPATH = src:../headers 如果定义了VPATH，那么，make就会在当当前目录找不到的情况下，到所指定的目录中去找寻文件了。

上面的的定义指定两个目录，“src”和“../headers”，make会按照这个顺序进行搜索。目录由“冒号”分隔。（当然，当前目录永远是最高优先搜索的地方）

$@:表示目标集

$<:所有的依赖目标集

$(filter %.o,$(files))表示调用Makefile的filter函数

大多数的C/C++编译器都支持一个“-M”的选项，即自动找寻源文件中包含的头文件，并生成一个依赖关系。例如，如果我们执行下面的命令：

cc -M main.c

如果你使用GNU的C/C++编译器，你得用“-MM”参数，不然，“-M”参数会把一些标准库的头文件也包含进来。

make会把其要执行的命令行在命令执行前输出到屏幕上。当我们用“@”字符在命令行前，那么，这个命令将不被make显示出来，最具代表性的例子是，我们用这个功能来像屏幕显示一些信息。如：

@echo 正在编译XXX模块......

当make执行时，会输出“正在编译XXX模块......”字串，但不会输出命令，如果没有“@”，那么，make将输出：

echo 正在编译XXX模块......

正在编译XXX模块......

subsystem:

cd subdir && $(MAKE)

其等价于：

subsystem:

$(MAKE) -C subdir

## 概述

### 组成

HPC Challenge benchmark具体包括7个基准测试程序：

1. HPL：Linpack TPP benchmark， 用于测试线性方程的浮点运算速率（MPI全局并行）；
2. DGEMM：测试双精度矩阵乘矩阵的浮点运算速率（单处理器和完全并行）；
3. STREAM：测试持续内存带宽（GB/s）和相应的矢量内核计算速度（单处理器和完全并行）；
4. PTRANS：并行矩阵变换，两两处理器间同步互相通信，测试网络通信能力（MPI全局并行）；
5. RandomAccess：测试内存整数随机更新的速率GUPS（单处理器、完全并行和全局并行）；
6. FFT：测试双精度复杂一维离散傅立叶变换DFT的浮点运算速率（单处理器、完全并行和全局并行）；
7. b\_eff：通信带宽和延迟，一系列MPI测试程序用于测试多种同步通信模式下的带宽和延迟。

HPCC提出了三种运行方式：Local（单处理器），Global（全局并行）和EP（完全并行）。Local方式表示程序在单个处理器上运行，Global方式在所有处理器上运行。EP方式也是在所有处理器上运行，但各处理器之间不进行通信。对这三种运行方式的结果进行比较，可以对系统的互连网络性能有进一步的了解。

### 规则

1. 基本运行

允许以下优化：

1. 编译和加载选项：允许使用提供商支持的编译和加载标志，包括移植，优化，调研预处理器。
2. 库：允许连接BLAS,FFT,MPI优化库，但要遵守以下规则：1、所有库必须完全披露，提供库名，版本，提供源码的机构；2、非一般库不许使用，除非报告结果的6个月内可获得；3、调用库须与发布的基准程序代码具有相同语义和语法，不允许不同库具有不同调用格式。
3. 软件工具：任何运行基准程序的工具（包括预处理器，编译器，静态和动态链接，操作系统）必须在报告结果的6个月内保持不变。
4. 必须提供完整输出结果，不接受部分结果。
5. 优化运行
6. 库：根据要求，BLAS,FFT,MPI这些库应该能被其他人使用。
7. 代码修改：以下程序可替换：1、HPL：HPL\_pdgesv()；2、DGEMM没有可替换的程序；3、PTRANS：pdtrans()；4、STREAM：tuned\_STREAM\_Copy(),tuned\_STREAM\_Scale(),tuned\_STREAM\_Add(),tuned\_STREAM\_Triad()；5、RandomAccess: Power2NodesMPIRandomAccessUpdate(), AnyNodesMPIRandomAccessUpdate(), and RandomAccessUpdate() 4、FFTE: fftw\_malloc(), fftw\_free(), fftw\_create\_plan(), fftw\_one(), fftw\_destroy\_plan(), fftw\_mpi\_create\_plan(), fftw\_mpi\_local\_sizes(), fftw\_mpi(), fftw\_mpi\_destroy\_plan()
8. 优化限制：1、计算必须在完全精度下执行（64位），能采用其他替代算法，但必须遵循规则2；2、替换算法必须完全披露，并交给HPC组织检查。替代算法必须与基准算法一样健壮，对矩阵相乘算法，不能用Strassen算法，因此它改变了执行次数；4、不允许使用任何解决方案的知识；5、不允许任何形式的规避实际计算。
9. 必须提供完整输出结果，不接受部分结果。

### 相关词汇

* BLAS：Basic Linear Algebra Subprograms，将矢量和矩阵存储在连续内存空间的基本操作。主要HPC平台的供应商和非供应商版本有ATLAS和Got BLAS。
* FFTE：快速傅立叶变换包，计算1，2，3维复杂双精度序列的离散傅立叶变换，支持的根有2，3，4，5和8。
* PARKBENCH：PARallel Kernels and BENCHmarks，包含CPU，内存，通信基准方程。PTRANS（parralel matrix transpose，并行矩阵转置）是其中一个。
* TOP500：TOP500始于1993年，提供高性能计算领域发展趋势的可靠跟踪和测试。一年2次，收集最top500的超级计算机。
* Effective Bandwidth Benchmark：b\_eff测试并行或分布计算机系统的累计通信网络带宽。采用不同消息大小，通信模式和方法。算法考虑真实应用中最短和最长消息在不同带宽下的传输平均值。
* High Performance Linkpack（HPL）：在分布式内存机器下密集线性方程的双精度（64位）计算。被认为是可移植且免费的the High Performance Computing Linpack Benchmark。
* STREAM Benchmark：测量持续内存带宽（MB/s）和相应的简单矢量内核的计算速率。
* TopCrunch：跟踪高性能计算机系统和软件工程的综合性能趋势。

## 输入文件

Hpccinf.txt为输入文件，在根目录下有\_hpccinf.txt的示例文件。hpl/www/tuning.html介绍HPL输入文件的设置。

\_hpccinf.txt示例：

1 HPLinpack benchmark input file

2 Innovative Computing Laboratory, University of Tennessee

3 HPL.out output file name (if any)

4 8 device out (6=stdout,7=stderr,file)

5 1 # of problems sizes (N)

6 1000 Ns

7 1 # of NBs

8 0 NBs

9 0 PMAP process mapping (0=Row-,1=Column-major)

10 1 # of process grids (P x Q)

11 2 Ps

12 2 Qs

13 16.0 threshold

14 1 # of panel fact

15 2 PFACTs (0=left, 1=Crout, 2=Right)

16 1 # of recursive stopping criterium

17 4 NBMINs (>= 1)

18 1 # of panels in recursion

19 2 NDIVs

20 1 # of recursive panel fact.

21 1 RFACTs (0=left, 1=Crout, 2=Right)

22 1 # of broadcast

23 1 BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)

24 1 # of lookahead depth

25 1 DEPTHs (>=0)

26 2 SWAP (0=bin-exch,1=long,2=mix)

27 64 swapping threshold

28 0 L1 in (0=transposed,1=no-transposed) form

29 0 U in (0=transposed,1=no-transposed) form

30 1 Equilibration (0=no,1=yes)

31 8 memory alignment in double (> 0)

32 ##### This line (no. 32) is ignored (it serves as a separator). ######

33 0 Number of additional problem sizes for PTRANS

34 1200 10000 30000 values of N

35 0 number of additional blocking sizes for PTRANS

36 40 9 8 13 13 20 16 32 64 values of NB

* **Line** **1**: 注释说明行，默认为HPL Linpack benchmark input file。
* **Line 2**: 注释说明行，默认为Innovative Computing Laboratory, University of Tennessee。
* **Line 3**: 用户可选择是否重定向输出。首先是文件名，然后指明用途。默认为HPL.out output file name (if any)。这句话是指如果重定向输出，则文件名为HPL.out，其他部分为说明。
* **Line 4**: 指明输出文件，此行须按格式要求，以正数开头，其他部分为说明。

"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"为其它值时，测试结果输出至第三行所指定的文件中

默认值为6 device out (6=stdout,7=stderr,file)。

* **Line 5**:指明问题规模的大小，必须小于等于20，第一个整数重要，其他部分为说明。

3 # of problems sizes (N)

是指用户希望运行3个问题规模，问题规模由line6说明。

**问题1：问题规模N应该多大？**

为达到最高性能，问题规模应尽可能利用全部内存。HPL所需的内存量即为系数矩阵大小。例如，如果有4个节点，其内存都为256Mb，即总共有1Gb（125MB）。开根号后为11585。一方面需要留一些内存给OS，因此问题规模适合为10000。经验法则，总内存量的80%总是很好的尝试。如果问题规模过大，页面交换将降低效率。

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%。

这只是一个参考值，具体N最优选择还跟实际的软硬件环境密切相关。当整个系统规模较小、节点数较少、每个节点的内存较大时，N可以选择大一点。当整个系统规模较大、节点数较多、每个节点的内存较小时是，N可以选择大一点。

* **Line 6**: 指明问题规模，line 5如果为3，则使用前3个正数，其余忽略。

3000 6000 10000 Ns

是指用户希望xhpl运行前3个问题规模，分别为3000，6000，10000。

* **Line 7**: 指明求解矩阵分块大小的个数，必须小于等于20，第一整数重要，其他部分为说明。

5 # of NBs

是指用户希望采用5个块大小，块大小由line8说明。

* **Line 8**: 指明块大小，line7如果为5，则使用前5个正数，其余忽略。

80 100 120 140 160 NBs

是指xhpl使用5个块大小，分别为80，100，120，140，160。

**问题2：块大小NB应该多大？**

NB用于数据分布和计算粒度。从数据分布角度，NB越小，负载越平衡。从计算粒度角度，过小的NB会导致通信增加，从而限制效率。高效的矩阵乘一般是内部分块。好的块大小最起码应在[32 .. 256]之间，最佳值取决于计算/通信效率比。如果根据效率44为一个好的块大小，当问题规模增大时，88或132很可能获得较好结果。

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。

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。

|  |  |  |  |
| --- | --- | --- | --- |
| 平台 | L2 Cache | 数学库 | NB |
| Intel P4 Xeon | 512KB | ATLAS | 400 |
| MKL | 384 |
| GOTO | 192 |
| AMD Opteron | 1MB | GOTO | 232 |

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。

* **Line 9**: 指明MPI进程如何映射到节点，即处理器阵列是按列还是按行的排列方式。推荐按行。

0 PMAP process mapping (0=Row-,1=Column-major)

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。

在HPL文档中，其建议采用按行的排列方式，我不理解其原因，可能和MPI任务递交的不同方式有关吧。

* **Line 10**: 指明进程网格的个数，必须小于等于20。第一个整数重要，其他部分为说明。

2 # of process grids (P x Q)

是指使用2个进程网格，网格由line11说明。

* **Line 11/12**: 指明每个网格的处理器行列数，例如line10为2，则使用每行的头两个正整数，其余忽略。

1 2 Ps

6 8 Qs

是指使用2个进程网格，分别是1×6和2×8，因此最少需要启动16个进程（取Pi×Qi的最大值），这两个网格将连续运行。如果启动大于16个进程，例如52个，那么只有6个（1×6）用于第一个网格，然后16个（2×8）用于第二个网格。HPL不会用完52个进程，而只用16个，因此该行应设为

4 2 Ps

13 8 Qs

**问题3：进程网格P\*Q应该是多少？**

取决于网络内部的物理连接。P和Q应相近，且Q略大于P。例如2×2，2×4，2×5，3×4，4×4，4×6，5×6，4×8…如果采用以太网，只有一条交换链路，将会严重限制HPL的效率和划分，适合采用较简单的网格，例如1×4，1×8，2×4…避免采用1×Q或P×1的网格。

二维处理器网格（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。对于Intel Xeon来说，关闭超线程可以提高HPL性能。
* P≤Q；一般来说，P的值尽量取得小一点，因为列向通信量（通信次数和通信数据量）要远大于横向通信。
* P = 2n，即P最好选择2的幂，P=1,2,4,8,16,…。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。例如，当系统进程数为4的时候，P × Q选择为1 × 4的效果要比选择2 × 2好一些。另外，U的广播中，Long法和二元交换法也在P为2的幂时性能最优。
* 当进程数为平方数时，如进程数为64，试试4×16的方式，兴许性能要不8×8好。
* **Line 13**: 指明残差阈值。第一个实数重要，其他部分为说明。

16.0 threshold

16.0一般适合所有情况。一些情况残差会大些，如35.6，xhpl会标记所有运行错误，但实际是正确的，但如果大很多，如10^6，则肯定是错的。如果设为0.0，则所有运行将认为是错的。如果设为负值，则略过对比，从而节约时间。

-16.0 threshold

这个值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。

以下为算法特性设置部分，xhpl将运行各种问题规模，块大小，进程网格的组合。因式分解采用RFACT（recursive panel fact），当因式小于NBMIN列时递归停止，经典递归采用NDIV=2，NBMIN=1。有三种LU分解算法的等价算法（left-looking，Crout，right-looking），HPL可选择任意组合用于RFACT和PFACT。

* **Line 14-21**: 指明LU分解方式。

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)

在消元过程中，HPL采用每次完成NB列的消元，然后更新后面的矩阵。这NB的消元就是L的分解。每次L的分解只在一列处理器中完成。

对每一个小矩阵作消元时，都有三种算法：L、R、C，分别代表Left、Right和Crout。在LU分解中，具体的算法很多，HPL采用这三种。

HPL中，L分解采用递归的方法，其伪代码如下：

**nn=NB;**

**Function L分解（nn）**

**{**

**if nn≤NBMINs 递归的L分解(nn); （NBMINs由第17行指定）**

**将nb分为NDIVs部分；**

**for （NDIVs次）**

**对每一部分作 L分解（nn/NDIVS） （DIVs由第19行指定）**

**对整个部分采用RFACTs算法作消元（RFACTs第21行指定）**

**}**

**Function递归的L分解(nn)**

**{**

**用PFACTs算法对nn列作消元（PFACTs由第15行指定）**

**}**

NDIVs选择2比较理想，NBMINs 4或8都不错。而对于RFACTs和PFACTs，好像对性能的不大（在LU消元算法中，说Crout算法的性能不错）。我们对这两个参数作了大量的测试，没有发现什么规律，可能它是由于它对性能的影响较小，使得这个性能的微小差别由于机器的随机性而无法区分。

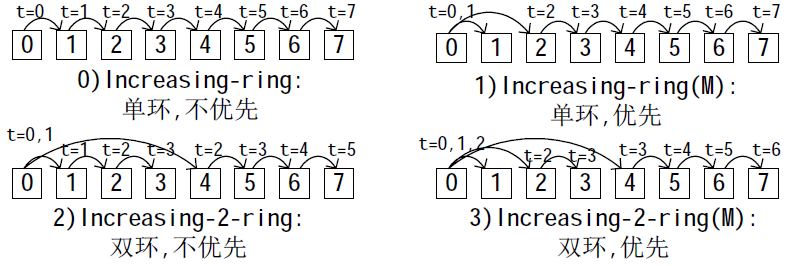
* **Line 22/23**：指明L的横向广播方式。为算法主循环，列广播。好的通信取决于问题规模和硬件效率。

2 # of broadcast

1 3 BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)

HPL中共提供六种广播方式。其中前四种适合于快速网络；后两种采用将数据切割后传送的方式，主要适合于速度较慢的网络。目前，机群系统一般采用千兆以太网甚至Myrinet、Infiniband、SCI等高速网络，所以一般不采用后两种方式。

前四种算法如图所示，分别采用单环/双环、第一列处理器不优先/优先。



对于系统规模较小、处理器数（进程数）较少的系统来说，这四个选择对性能影响很小。

对于横向处理器数Q较大的网络来来说，选择双环可以减少横向通信宽度，较小横向通信延迟。另外，第一列处理器优先算法也可以确保下一次L分解的尽早开始。

根据测试经验，在小规模系统中，一般选择0或1；对于大规模系统，3是个不错的选择。

* **Line 24/25**: 指明横向通信的通信深度。0和1一般是较好的选择，但仍取决于问题规模和机器配置。

2 # of lookahead depth

0 1 DEPTHs (>=0)

0说明在当前分解完全完成并更新后开始下一次分解；1说明更新后立刻开始下一次分解；k说明更新后立刻开始下k次的分解。1一般能获得好结果，但需要大规模计算。如果不懂的话用1，否则用0。

L的分解过程是一个相对比较耗时的过程，为了提高性能，其采用先作一部分分解，然后将这一部分结果广播出去。“DEPTHs”值就是说明将L分几次广播。DEPTHs＝0表明将L一次性广播出去，也就是将整个L分解完成以后在一次性广播；DEPTHs＝1表示将L分两次广播；依此类推。

L分为多次广播可以使得下一列处理器尽早得到数据，尽早开始下一步分解。但这样会带来额外的系统开销和内存开销。DEPTHs的值每增加1，每个进程需要多申请约（N/Q＋N/P＋NB＋1）×NB×8的内存。这对HPL的开销是很大的，因为增加DEPTHs以后，为了保证不使用缓冲区，不得不减小问题规模N的值，所以在N和DEPTHs需要作一个权衡。

根据测试经验，在小规模系统中，DEPTHs一般选择1或2；对于大规模系统，选择2～5之间。

* **Line 26/27**：指明U的广播算法。有两种交换算法，一种是基于“binary exchange”，一种是基于“spread-roll”（即long）。对大规模问题，后一种效率更高。还可选择mix两种，当阈值大于列数，则采用“binary exchange”，否则采用“spread-roll”。阈值在line27设定。

2 SWAP (0=bin-exch,1=long,2=mix)

60 swapping threshold

是指HPL在大于60列的部分采用“spread-roll”，否则采用“binary exchange”。

Long和mix的效率区别取决于阈值相对NB的大小。如果阈值很大，则会一直采用bin-exch，如果阈值小于NB，则会一直采用long。

U的广播为列向广播，HPL共提供了三种U的广播算法：二元交换（Binary Exchange）法、Long法和二者混合法。SWAP=“0”，采用二元交换法；SWAP=“1”，采用Long法；SWAP=“2”，采用混合法。

二元交换法的通信开销为log2P×（Latency＋NB×LocQ（N）/Bandwith），适用于通信量较小的情况；Long法的通信开销为（log2P＋P－1）×Latency＋K×NB×LocQ（N）/Bandwith，适用于通信量较大的情况。其中P为列向处理器数，Latency为网络延迟，Bandwith为网络带宽，K为常数，其经验值约为2.4。LocQ（N）＝NB×NN为通信量，NN随着求解过程的进行逐步减少。

由于NN在求解过程中在不断的变化，为了充分发挥两种算法的优势，HPL提供了混合法，当NN≤swapping threshold（第27行指定）时，采用二元交换；否则采用Long法。

一般来说，我们选择混合法，阈值可通过公式求得一个大概值。对于小规模系统来说，此值性能影响不大，采用其缺省值即可。

* **Line 28/29**：指明L和U的数据存放格式。

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

是指上三角矩阵存为转置形式。

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

是指行U存放为转置形式。

line28选哪个并不重要，如果不懂则选0。line29控制行存储方式，推荐0。

C语言矩阵在内存是按行存放的，Fortran语言是按列存放的。由于HPL采用C语言书写，而调用的BLAS库有可能采用C语言，也有可能采用Fortran语言编写。

若选择“transposed”，则采用按列存放，否则按行存放。

* **Line 30**：采用或不采用平衡阶段，在line26中选择1或2才会用到。

1 Equilibration (0=no,1=yes)

由于不知道如何生成随机矩阵且开销较小，可以采用。

* **Line 31**：指明HPL在内存分配中地址对齐方式。一般采用4，8，16，可能导致一些内存浪费。

8 memory alignment in double (> 0)

## 其他优化

1. MPI

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”。

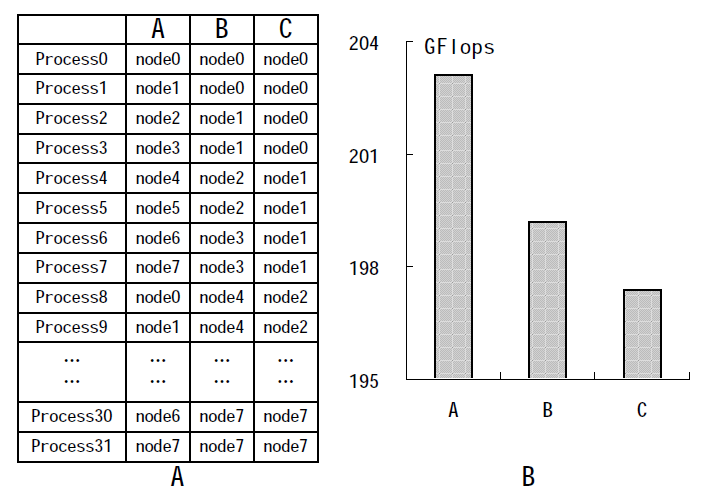
对于GM来说，在找到路由以后，将每个节点的gm\_mapper进程kill掉，大概有一个百分点的性能提高。当然也可以采用指定路由表的方式启动GM。

1. BLAS库的选取

BLAS库的选取对最终的性能有着密切的关系，选取合适的BLAS库是取得好性能的关键。目前BLAS库有很多，有Atlas、GOTO、MKL、ACML、ESSL等等。根据我的测试经验，其性能是在Xeon和Opteron平台上，GOTO库性能最优。

1. 处理器－进程的映射方式

调节进程与处理器间的映射关系对性能产生不小的影响，优化此映射关系的关键在于改变各节点的计算负载和通信操作以减少通信网络的竞争、实现更快速的通讯路径和实现节点的计算负载均衡。如：避免计算负载过于集中于某几个节点、避免两节点间同时多对进程并发通信、尽可能使用节点内通信等等。



在四路SMP Cluster系统中，进程与处理器的映射关系主要有三种排列方式，如图所示。A方式为顺序排列进程与处理器间映射关系；C方式使得相邻进程间通信通过节点内通信实现；B方式介于两者之间。HPL进程与二维处理器网格之间采用列优先的映射关系。

考虑HPL计算和通信最密集的PANEL分解，PANEL分解采用“计算—通信—计算”的模式，其中通信是采用二元交换法交换主元所在行。列中所有的进程都参与每一次通信，通信的并行度很高且并发执行。A方式中，每一列进程中没有两个进程在同一节点上，列进程间通信都是节点间通信，但计算分布在32节点上，计算负载更为均衡，且不会出现两节点间多对进程同时通信、抢占同一通信网络的情况。C方式中，每一列进程中每四个进程在同一节点上，此四进程间通信通过节点间通信完成，但是C方式下会出现两节点间两对或四对进程同时并发通信、抢占同一通信网络的情况，且每一次PANEL分解集中在八个节点上，此时这八个四个CPU同时工作，其余节点都在等待，计算负载极不均衡。

图B是三种不同的映射关系在D4000A八个节点上的测试结果，A方式的性能最优，B次之，C最差。

1. 操作系统

操作系统层上的性能优化方法很多，如裁减内核、改变页面大小、调整改内核参数、调整网络参数等等。将一些没有必要的系统守护进程去掉，并且将操作系统启动到第3级，不要进入图形方式。

1. 编译优化

采用编译优化可以在很大程度上提高CPU密集型应用程序的性能，如采用超长指令可以较大程度的提高浮点数处理性能。编译优化在不修改程序的条件下主要有两种方法：采用性能较好的编译器和增加编译优化选项。在X86平台上，主要编译器有GNU、Intel编译器、PGI编译器等，在一些专门的平台上有专门的编译器，如IBM的P系列机器上有其专有的编译器xlc和xlf。编译优化选项和编译器密切相关，不同的编译器编译优化选项不尽相同。

在HPL测试中，编译优化对其性能影响不大。原因是HPL将计算最密集部分的都通过调用BLAS库完成，在HPL本身的程序中，作数值计算的几乎没有。

1. 其它硬件设备对性能的影响

我这里说的其它硬件设备是指除了CPU以外的设备，包括网络、内存、主板等等。虽然HPL主要测试CPU的性能，但是计算机是一个整体，其它的硬件设备对其影响也是很大。

先说网络，网络是机群系统的核心。当然网络性能越好，整体性能越好。但是对于同一种网络，如千兆以太网，网线的连接等也会对性能造成影响。首先要了解所使用的交换机的性能特点，同样是千兆以太网，其性能差别会很大，不同端口之间通信的速度不尽相同。还有就是主板和内存，其性能特点也会对整体性能有很大的影响。

1. 其它

对于Intel XEON CPU来说，关闭超线程可以得到更好的性能。对于大多数HPC应用程序来说，CPU占用率比较高，所以超线程技术很难发挥其优势。关闭超线程是一个很好的选择。

相关资料

<http://www.orientsky.com.cn/_new/Read.asp?ID=1495>

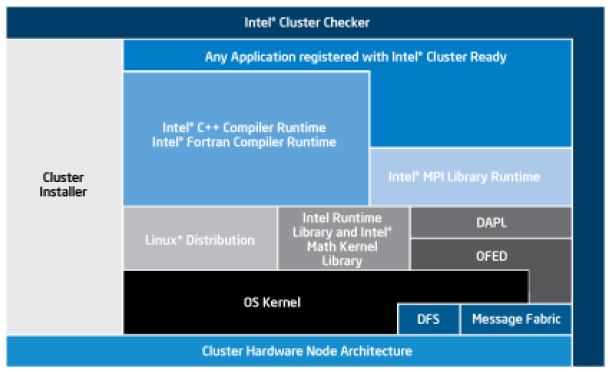
<http://www.orientsky.com.cn/_new/Read.asp?ID=1496>

# Intel Cluster Checker

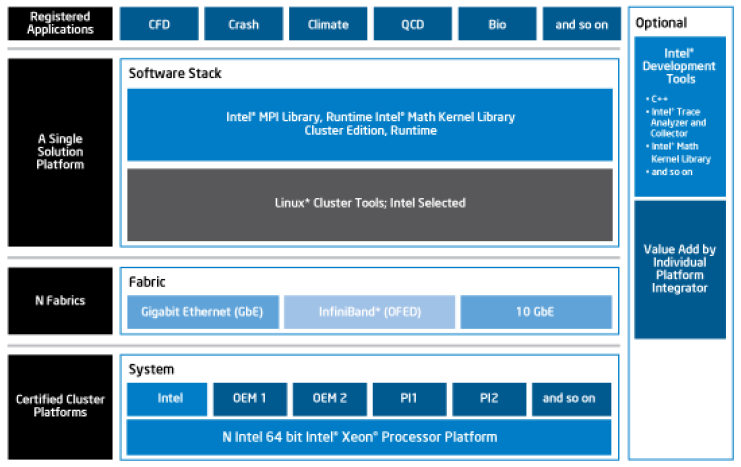
简言之，Intel Cluster Ready是一套标准，比如硬件要满足什么要求，要安装哪些软件，采用哪种系统架构等，满足该标准的集群能实现高性能计算。而Intel Cluster Checker是由120多个测试程序组成的一套工具集，用于测试集群的各项指标，比如节点间是否能ping通，gcc、mpi的版本是否相同等等，如果测试全部通过，则满足了Intel Cluster Ready标准，集群就没问题了。

## 概述

英特尔与戴尔和其他领先高性能计算(HPC)供应商共同开发了Intel Cluster Ready计划，来帮助简化购买、构建、部署和管理HPC群集(HPCC)，主要包括Intel Cluster Checker工具。

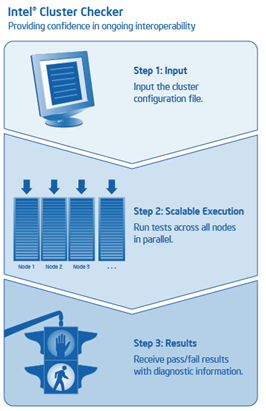


Intel Cluster Ready系统结构



Intel Cluster Checker是一个强大的集群诊断工具，可在无先前集群经验情况下使HPC获得高效。

* 平台提供商采用Intel Cluster Checker验证其系统完全符合Intel Cluster Ready的要求。软件开发者采用Intel Cluster Checker证明其开发的软件能成功运行在Intel Cluster Ready系统上。
* 采用Intel Cluster Checker帮助所有组件合作运行。定期运行Intel Cluster Checker有助于增强系统可靠性并确保最佳性能。
* 无论是软件更新引起了软件冲突，还是电缆连接松动，Intel Cluster Checker都能迅速确定问题并提供详细的诊断信息。使用Intel Cluster Checker帮助减少排除故障所用的时间，并最小化对专业支持技能的需求。
* Intel Cluster Checker分析集群配置，以符合Intel Cluster Reader要求。当增加新功能时，运行Intel Cluster Checker以保证系统仍然符合接口要求。
* Intel Cluster Checker包含多于100项检查，工具集是可扩展的，用户可以通过指定命令和预期输出，甚至通过插件程序与其他测试集成，来自定义测试。Intel Cluster Checker检查集群和节点，还涉及防火墙，内核，存储，网络设置等。采用Intel MPI基准程序，STREAM，HPCC和其他基准程序测试性能。
* 在工业设计阶段采用Intel Cluster Checker以保证方案达到Intel Cluster Ready要求。在制造阶段采用Intel Cluster Checker保证集群功能达到预期，尽可能发现问题。



Intel Cluster Checker验证流程

## 用户手册

### 执行步骤

Intel Cluster Checker包含一系列独立测试模块，每个模块完成单一节点或整个集群的某一特定测试。遵循以下步骤：

1. **第一步，建立运行时环境。**

执行脚本clckvars.sh，保存在安装目录下。

$ source <installpath>/clckvars.sh

初始化脚本功能包括：

* 增加cluster-checker到执行路径。
* 命令行选项能用tab补充完整。
* 能man用法。$ man clustercheck
* 查询帮助文档的命令格式为：clck\_<test\_module\_name>（以hardware\_uniformity为例）$ man clckhardware\_uniformity

1. **第二步，生成节点文件**

Nodes file包含节点主机名或ip，每个一行。例如计算集群叫cluster，包含四个节点。

# /home/icr/nodesfile: Cluster nodes to process

node1

node2

node3

node4

1. **第三步，生成配置文件。**

配置文件为xml格式，保存Intel Cluster Checker运行时行为。最简单的配置文件只包含<nodefile>元素。

<!Myconfiguration file: /home/icr/myconfig.xml >

<cluster>

<nodefile>/home/icr/nodesfile</nodefile>

</cluster>

配置文件还可包含测试模块的配置，例如测试哪些模块和它们的常用参数。如果没有提供测试模块，Intel Cluster Checker将运行预定义的模块。

1. **第四步，运行Intel Cluster Checker。**

调用cluster-check，并传递配置文件。

$ clustercheck myconfig.xml

Intel Cluster Checker依次执行相关模块，输出结果。

如果没有提供配置文件，cluster-checker会在以下路径中搜索，采用最早找到的配置文件：

1. <install\_path>/etc/config.xml

2. /etc/intel/clck/config.xml

1. **第五步，分析输出**

Intel Cluster Checker生成2个输出文件，其文件名与配置文件名相同，再加上时间戳和特定后缀。

|  |  |
| --- | --- |
| File name | Description |
| myconfig20110304.085149.out | .out为控制台输出 |
| myconfig20110304.085149.xml | .xml为Intel Cluster Checker输出的全部内容，可用工具解析。 |

1. **第六步，重新执行。**

修改nodes文件，配置文件或集群配置，重新从第二步开始执行，直到没有警告。

### 配置

节点分为三种类型：compute，head和other。一些测试项只测试特定类型节点，其他根据节点类型操作不同。

#### 节点文件

节点定义在文本文件中。

# list of nodes to check

node1

node2

node3 # fails intermittently

node4

#可用于注释，当后接”type:”或”group:”时，指明节点类型。例如”#type: head”，节点为主节点，compute和other类似。如果没指明则默认为compute。如下所示，node1为主节点，node2和node3为计算节点，node4为其他。

# list of nodes to check

node1 # type: head

node2

node3 # type: compute fails intermittently

node4 # type: other

当节点有多个功能时，用“：”隔开。如下所示，node1为主/计算节点，node2/node3为计算/其他节点，node4为主/计算/其他节点。

# list of nodes to check

node1 # type: head type: compute

node2 # type: compute, other

node3 # type: compute type: other fails intermittently

node4 # type: other, head type: compute

主节点可直接跟在#后面。

node1 # head

可通过group标签对节点分类。如下所示，node1为计算/主节点，属于大内存组，node2拥有大内存和快速cpu，node3和node4为标准配置。光分组没用，必须在xml配置文件中使用相应的组才有用。

# list of nodes to check

node1 # type: compute, head group: bigmem

node2 # group: bigmem group: fastcpu

node3 # fails intermittently

node4

#### 配置文件

Xml配置文件例子在<installationpath>/examples/目录中。

执行前，Intel Cluster Checker XML解析器会验证参数是否正确，可用—force取消验证（不推荐）。

Intel Cluster Checker采用的格式为W3C XML(clck.xsd)和XLST(clck.xsl)配置文件须符合schema的要求，才可用xmllint工具验证参数是否正确，xmllint包含在libxml2包中。

首先调用xsltproc：

xsltproc output examples/sorted.xml clck.xsl examples/example.xml

然后调用xmllint：

xmllint schema clck.xsd examples/sorted noout

nodes file可通过XML配置文件读取，也可在命令行中通过—ndefile指明。

1. **节点设置**

<nodefile> file </nodefile>

如果file以$开头，则读取其后环境变量的值，例如PBS\_NODEFILE：

<nodefile>$PBS\_NODEFILE</nodefile>

<head> *value* </head>指明头节点的另一种方法，value必须在nodefile中。

<mixedhead/>标签，指明所有头节点同时也是计算节点。

<node\_suffix> *value* </node\_suffix>value加到集群名称后面，如mycluster1.mydomain。如果后缀为域名。则必须指明“.”。

1. **运行时行为设置**

<alltoallthreshold>*value* </alltoallthreshold>value（默认值为64）控制节点数增长，只当节点数小于或等于value时，all to all检测所有节点对。

<alltoallthrottle>*value* </alltoallthrottle>当节点数大于alltoallthreshold时，value值控制生成对数，相邻节点由nodefile中节点位置决定，而不是物理位置。默认值为1。

<head\_tempdir> *value* </head\_tempdir>生成主节点执行检查的临时目录/文件的默认路径，默认值为/tmp。必须是绝对路径，且目录权限为可读可写可执行。也可通过环境变量CLCK\_HEAD\_TEMPDIR设置。环境变量优先级高于配置文件。

<node\_tempdir> *value* </node\_tempdir>生成节点测试的临时目录/文件的默认路径。默认值为/tmp。必须是绝对路径，目录权限是可读可写可执行。也可通过环境变量CLCK\_NODE\_TEMPDIR设置。环境变量优先级高于配置文件。

<processlimit>*value* </processlimit>同时检测的节点个数（每个节点生成一个进程并行检测）。进程个数不大于value值，默认值为64。

<retry> *value* </retry>检测失败后重新执行次数。

<user> *value* </user>执行测试的系统用户名。只影响那些常规用户调用特权用户才能使用的工具的模块，如passwd。

<env> *export NAME=VALUE* </env>设置用户自定义的环境变量，可重复定义以设置多个环境变量。该标签可用于全局或模块内部。如果全局变量在模块内部重新设置，则被重写为模块变量。

<cluster>

<env> export HOME=/home/$USER </env>

<nodefile>/etc/intel/clcknodelist</nodefile>

<test>

<intel\_mpi\_rt>

<env> export I\_MPI\_PERHOST=1 </env>

<env> export I\_MPI\_DEBUG=5 </env>

<device>rdssm</device>

<mpipath>/opt/intel/mpirt/3.1</mpipath>

<processnumber>2</processnumber>

</intel\_mpi\_rt>

</test>

</cluster>

环境变量按声明顺序设置，因此如果有依赖关系，则注意声明顺序。Cshell用set替代export。

1. **设置测试模块**

可通过配置文件设置相应模块标签或命令行选项修改需测试的模块。

<exclude\_module> *test\_module* </exclude\_module>单独运行该模块。如果该模块依赖于其他模块，其他模块也会被执行。

*<include\_module> test\_module </include\_module>*在标准测试以外运行该模块。如果该模块依赖于其他模块，其他模块也会被执行。

<include\_only\_module> *test\_module* </include\_only\_module>忽略默认模块而只运行该模块。该选项优先于<include\_only>，将忽略<include\_only>的设置。

每个模块有一个执行等级，可通过—level修改。

除了默认模块，还有几个预定义集合可通过命令行设置，如—compliance,--sdk-compliance,--certification和—deployment，它们会修改测试内容。

1. **配置测试模块**

测试模块配置在<test>…<test>标签中，模块参数包含在以模块名命名的标签中。

<cluster>

...

<test>

...

<clock\_sync>

<deviation>30</deviation>

</clock\_sync>

...

</test>

</cluster>

各模块具体参数参看Intel Cluster Checker Test Module Reference。

### 运行测试

Clustercheck *xml\_config\_file options*

Intel Cluster Checker将结果同时输出到终端，2个以时间戳命名的输出文件中（位于当前目录下）。.out即为终端内容，.xml为全部输出。

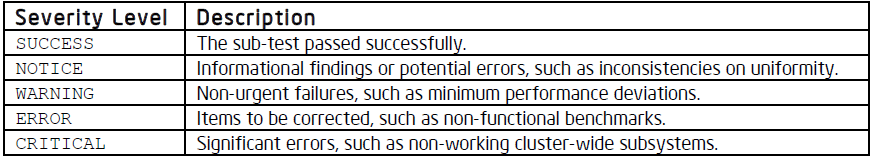
以下模块需要超级用户权限。如果普通用户运行，则会跳过该测试。因此推荐针对Cluster Checker设置特定用户以保存输出结果。环境变量TEMP用于定义程序解压临时库的位置，默认为/tmp，相同的还有CLCK\_HEAD\_TEMPDIR和CLCK\_NODE\_TEMPDIR。

TEMP=/home/icr clustercheck

1. **Console Output**

默认输出包含头和结果两部分，头部分显示信息包括命令行，用户凭证，执行开始时间等。结果部分显示所有测试模块，名字和描述。在默认输出等级下，只输出失败的测试。

根据问题的严重程度划分测试结果分别为SUCCESS,NOTICE,WARNING,ERROR,CRITICAL。



在基础测试中，如果ping、ssh出错，问题严重程度都为CRITICAL。在性能测试中，根据性能与期望的偏差决定，性能阈值设置在配置文件中，偏差超过5%为WARNING，20%为ERROR，50%以上为CRITICAL。如果没有设置阈值，则为NOTICE。

1. **Log Files**

日志输出路径的搜索采用回退机制，为第一个找到的权限允许的目录：

1. 环境变量CLCK\_LOG\_DIRECTORY设置的路径；
2. /var/log/intel/clck/
3. <installation\_directory>/logs/
4. 当前路径
5. **Additional Output**

<verbose>*value </verbose>*输出等级，value越大，输出内容越全面。

<debug/>设置该标签意味着生成一个名为testmodule.timestamp.debug的文件，其中包含每个节点的命令和相应输出。也可通过命令行中增加—debug设置。

1. **Command Line Options**

--autoconfigure:自动配置文件。

--compliance version:检查集群服从Intel Cluster Ready Specification第几版本的要求，默认为Versions 1.0.x。可用“：”分隔多个版本号，以在一次执行中检测多个版本。如果不提供则在/etc/intel/icr搜索，如果搜索不到版本文件，则检查所有版本。

--level value:每个模块有个等级。快测试等级低（1），慢测试等级高。--level告诉程序只允许等级小于或等于该value的测试。value最小为1，最大为5，默认为3。

--list:打印所有测试模块然后退出。

--nodefile:检测file中所列的节点。

--nolog:不输出文件。

。。。。。

1. **Environment Variables**

CLCK\_LOG\_DIRECTORY:输出结果到指定目录。

CLCK\_MODULE\_PATH:第三方测试模块的搜索路径，多个路径可用“；”隔开。

CLCK\_HEAD\_TEMPDIR:改变临时文件生成路径，默认为/tmp。路径权限必须可读可写可执行。也可通过xml配置文件中的<head\_tempdir>设置。环境变量优先级高于配置文件。

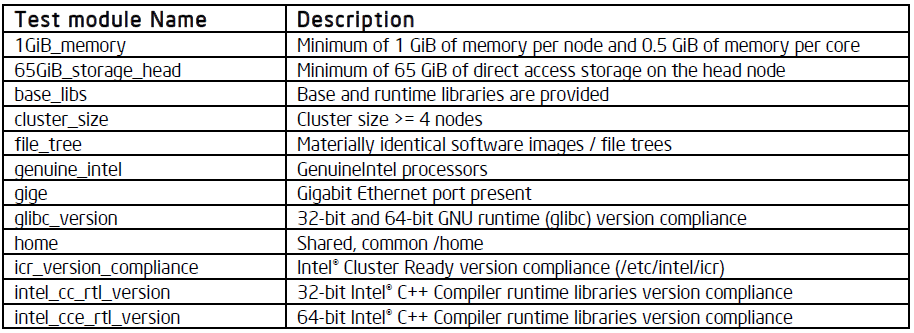
CLCK\_NODE\_TEMPDIR:改变临时文件生成路径，默认为/tmp。路径权限必须可读可写可执行。也可通过xml配置文件中的<node\_tempdir>设置。环境变量优先级高于配置文件。

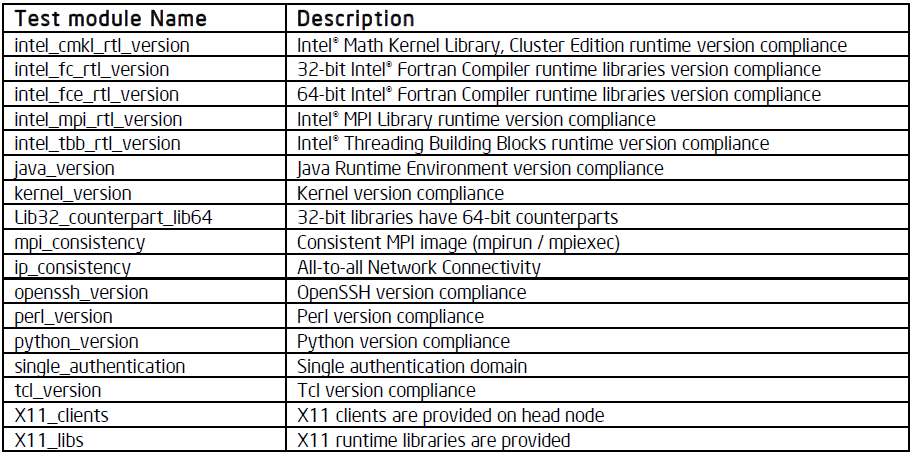
CLCK\_REGULAR\_USER:需要以超级用户执行某些模块的时候，普通用户的身份。

### 测试分类

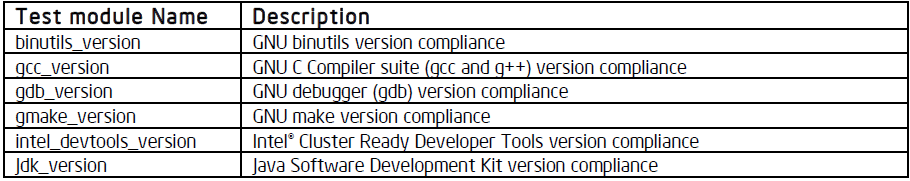
分为4类：

1. 检测是否符合基本的Intel Cluster Ready Specification。在命令行中用—compliance设置。
2. 检测是否符合Intel Cluster Ready Specification中Develper Cluster部分。在命令行中用—sdk-compliance设置。
3. 检测集群是否配置正确，能按预期执行。默认设置。
4. 可选测试。
5. **Compliance Test Module**





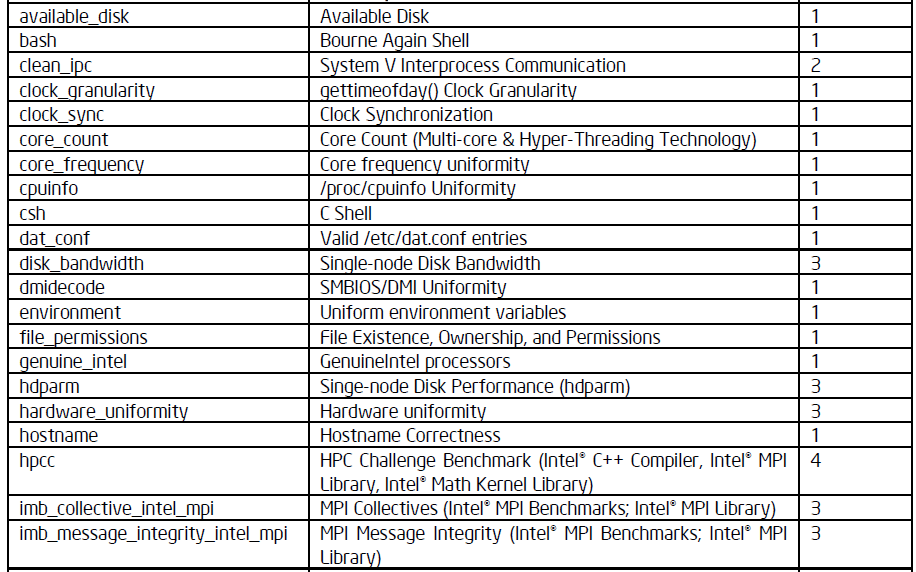
1. **SDK Compliance Test Modules**

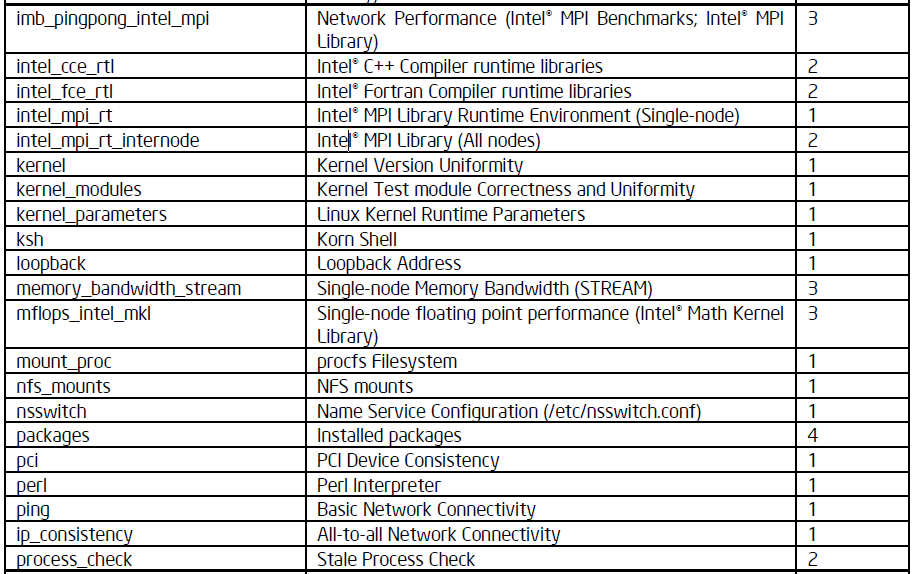


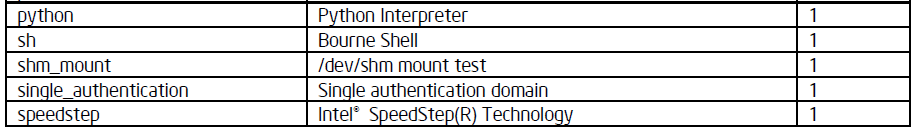
1. **Default Test Modules**

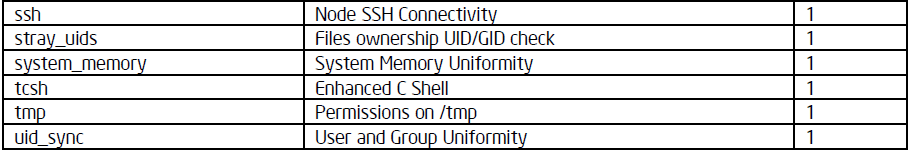
Intel Cluster Checker运行在wellness模式下，等级3。因此小于等级3的模块都被执行。通过—level修改等级可以从该集合中增加或删除测试模块。





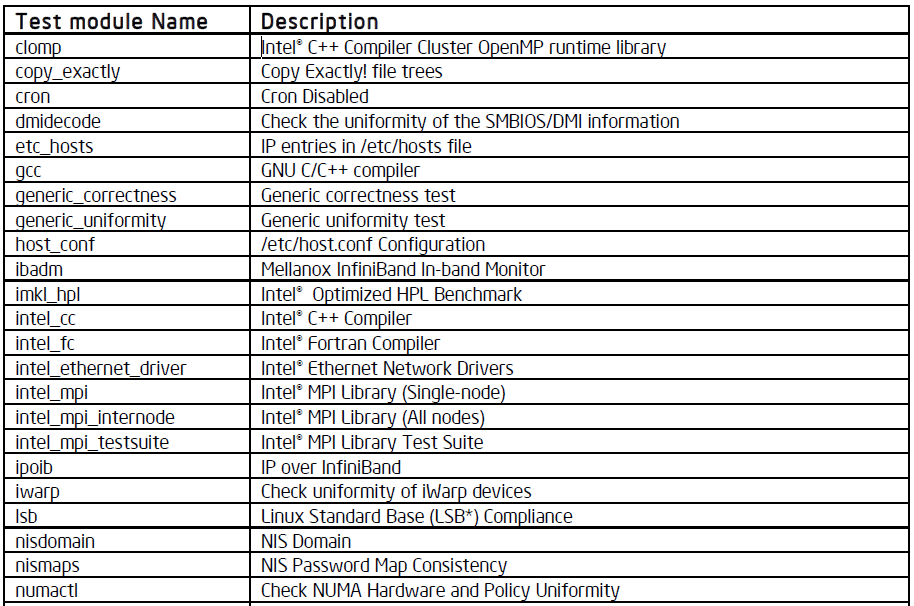


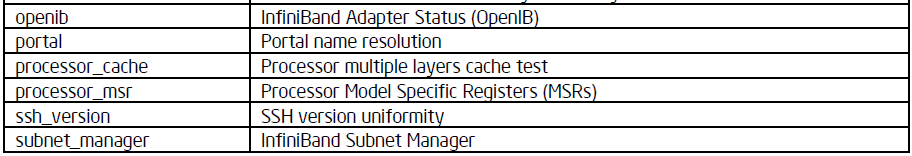




1. **Optional Test Modules**

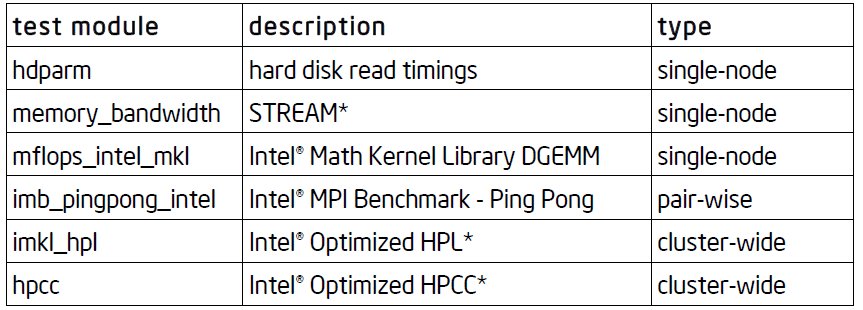
这些模块必须由用户指定才运行。





### 性能测试

采用以下模块测试集群或节点性能，分为single-node，pair-wise和cluster-wide三类：



1. **Single-node Benchmarks**

这类测试结果不依赖于集群质量，而是用来比较不同节点。大多数包含性能偏差测试，以检测所有节点拥有相似性能。性能偏差用中值和标准方差计算。允许范围为(median+/- factor\*stddev)，factor默认为3。

1. **Pair-wise Benchmarks**

这类测试选择集群中两个节点，测试通信效率。同样也与性能偏差以保证节点对间性能一致。

1. **Cluster-wise Benchmarks**

这类检测依赖于整个集群，如HPCC。

### 异构集群

Intel Cluster Checker通过group为节点分组。在node file中设置：

# list of nodes to check

node1 # head

node2 # group: XeonE5506

node3 # group: XeonX5560

node4

在XML 配置文件中设置，每个模块必须包含一个<group>标签：

<hardware\_uniformity>

<group name="XeonE5506"/>

<group name="XeonX5560"/>

</hardware\_uniformity>

1. **Sub-clusters**

内部划分子集群。对不同子集群采用不同配置。需在node file和配置文件中设置。

Node file中：

# list of nodes to check

node1 # head group: subCluster1

node2 # group: subCluster1

node3 # group: subCluster1

node4 # group: subCluster1

node5 # group: subCluster1

node6 # group: subCluster2

node7 # group: subCluster2

node8 # group: subCluster2

node9 # group: subCluster2

配置文件中：

<hpcc>

<group name="subCluster1">

<ccpath>/opt/intel/cce/9.1.038/</ccpath>

<fabric>

<bandwidth>.015</bandwidth>

<device>sock</device>

<dgemm>8.5</dgemm>

<fft>.5</fft>

<hpl>.023</hpl>

<latency>60</latency>

<ptrans>.15</ptrans>

<randomaccess>.002</randomaccess>

<stream>0.9</stream>

</fabric>

<mklpath>/opt/intel/cmkl/9.0/</mklpath>

<mpipath>/

opt/intel/impi/3.0/</mpipath>

<processnumber>4</processnumber>

<threadnumber>1</threadnumber>

</group>

<group name="subCluster2">

…

</group>

</hpcc>

1. **Fat Nodes**

是指拥有更高硬件的节点，如额外内存或二级缓存。

在node file中指明fat组：

# list of nodes to check

node1 # head

node2 # group: fat

node3

node4

在配置文件中定义相应参数：

<system\_memory>

<group name="fat">

<physical>8388608</physical>

</group>

<physical>4194304</physical>

<swap>4194304</swap>

</system\_memory>

### 自动配置

--autoconfigure简化初始配置。首先扫描节点以获取完成基本配置的信息，包括检测节点、配置路径，配置单节点性能测试模块的阈值等。

Autoconfigure [OPTIONS]，自动配置模式需要最基本的配置文件，因此必须通过命令行传递。其余参数分为两类：1是控制自动配置的目标；2是控制新配置文件的存储方式。选项间以“：”分隔，无空格。

1. **Autmatic Configuration Targets**

如果设置，则只配置设置目标，如果没设置，则采用默认值。

Global:采用统一配置设置Intel Cluster Runtimes Tools的路径，目标选项包括<cc-path>，<fc-path>，<mpi-path>，<mkl-path>。如果这些路径已经配置则会覆盖。这意味着这些工具必须按照Intel Cluster Ready Specification要求安装在所有节点。下例中，将根据统一路径配置而重新生成的配置文件，将位于输入文件同个目录下。

Clustercheck --auto global config.xml

Nodes:自动发现集群节点，生成新的node file。如果配置文件中包含node file的路径，或命令行中用—nodefile提供了路径，则不会自动搜索。下例中，新生成的node file位于配置文件目录下。

Clustercheck --auto nodes /home/icr/clck\_conf/config.xml

Performance：扫描集群计算节点，测试主要硬件设备并采用启发式计算获得单个节点的阈值提供给模块。目标模块为mflops\_intel\_mkl，memory\_bandwith\_stream等。下例中，程序尝试计算出阈值，然后对默认配置文件修改，生成新的配置文件，由于没有提供配置文件，程序在默认路径中搜索第一个找到的配置文件。

Clustercheck --auto performance

## 开发文档

### 模块分类

Intel Cluster Checker有四类测试模块：

Unit：检测节点属性是否正确，例如/tmp是否有正确的权限。

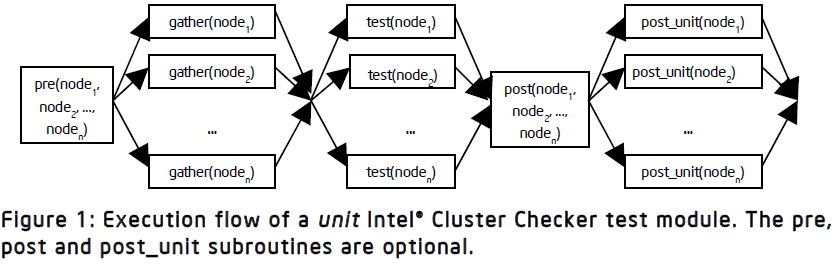
Vector：检测集群中节点配置是否统一，例如所有gcc版本相同。

Span：检测集群配置是否正确，例如MPI简单程序是否能正确运行。

Matrix：验证任意点到点配置是否正确，例如所有节点对间的网络延迟和带宽。

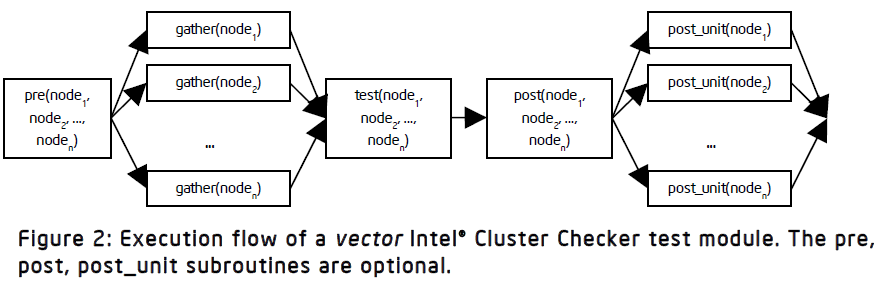
1. **Unit Class**

检测单个节点是否正确，即节点属性是否符合某个特定值，某个节点的正确性与其他节点无关。因此搜集和测试阶段可分别并行执行。



1. **Vector Class**

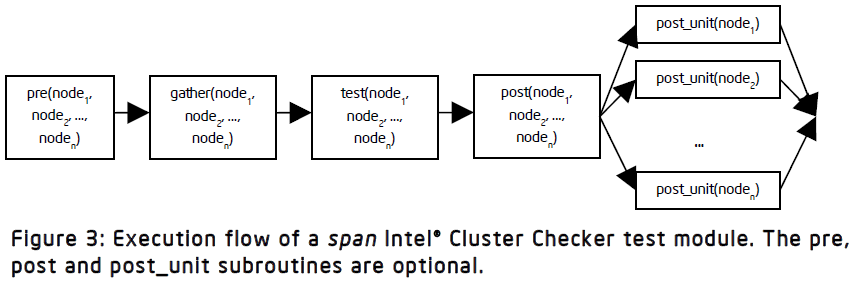
检测所有节点搜集信息是否相同。因此测试前需搜索所有节点的信息。搜索阶段可并行执行，测试阶段必须检查一系列值。



Vector比Unit更灵活，通过test阶段在所有节点中循环，unit可用vector实现，但效率较低。例如gcc检测可编译和运行hello world，同时验证所有节点的gcc版本相同。

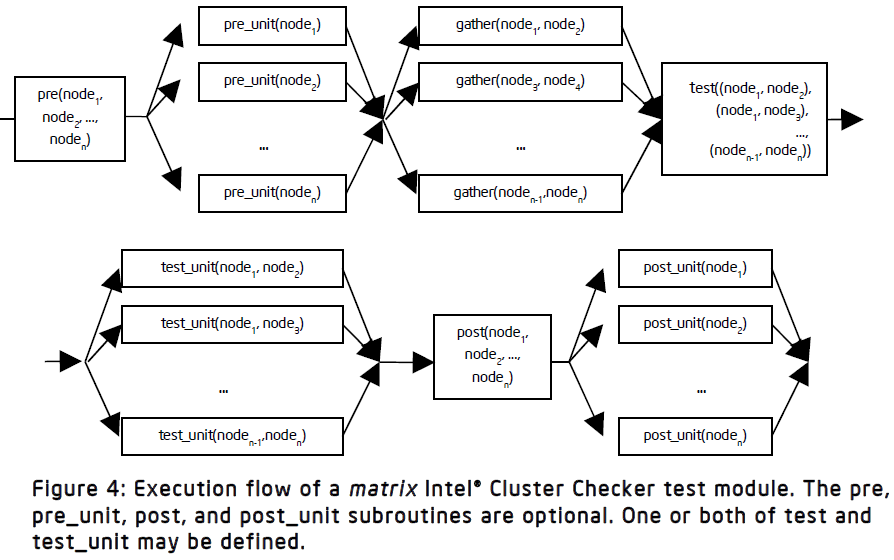
1. **Span Class**

检测集群配置是否正确，需要所有节点信息。



1. **Matrix Class**

检测节点对，收集阶段可并行。可通过<alltoall\_throttle>控制节点对个数。



### 编写模块API

外部模块必须用标准Perl编写，名字定义的格式为package name CLCK::Test::module\_name。程序在预定义路径中搜索模块，因此模块必须放在安装路径下的modules目录中，或CLCK\_MODULE\_PATH设置的路径中。

1. **Reserved Variables**

$check\_compute：在compute类型的节点上运行测试，默认值为true

$check\_dedicated\_head：在除了head类型的节点上运行测试。

$check\_head：在head类型的节点上运行测试，默认值为true。

$check\_other：在node类型的节点上运行测试，默认值为true。

$class：模块类型，可选值为unit，vector，span，matrix。

@depend：依赖的模块，如果依赖的模块失败，当前模块跳过。

$description：描述。

$level：模块等级。

$long\_check：如果设置，程序将打印消息告诉用户这个模块将运行很长时间。

$pod：包含Perl Plain Old Document的字符串。

$user：true或false，指明模块是否以普通用户运行，默认值为true。

$version：模块版本。

### 示例

dmesg.pm：

package CLCK::Test::dmesg //命名

our $VERSION = '1.0'; //申明变量

our $class = 'unit';

our @depend = qw(ssh);

our $description = 'Check Kernel messages log';

our $level = 3;

our $user = 1;

# collect the kernel messages //各节点调用gather方法收集信息

sub gather {

my ($node) = @ARG;

my %result;

my $output = remote\_cmd( $node, 'dmesg' ); //remote\_cmd执行’dmesg’，返回结果

$result{$node} = $output;

return \%result;

}

sub test { //test方法并行执行，分析收集到的信息

my ( $result, $configuration, $node ) = @ARG;

my @status;

my %errors\_found;

my %needs\_found;

my @failures = qw(error warning);

# Load needed words from config file

my @needs;

if ( defined $configuration->{'need'} ) {

if ( ref( $configuration->{'need'} ) =~ m{ARRAY}xms ) {

push @needs, @{ $configuration->{'need'} };

}

else {

push @needs, $configuration->{'need'};

}

}

# Loads failing words from config file //从XML配置文件中解析出配置参数，存入哈希表中，以name为关键字，分析结果是否正确，返回一个CLCK::Result::Message数组的引用，其中包含子测试结果，每个子测试生成一个Message，并设置其描述，值，严重程度和状态

if ( defined $configuration->{'fail'} ) {

if ( ref( $configuration->{'fail'} ) =~ m{ARRAY}xms ) {

push @failures, @{ $configuration->{'fail'} };

}

else {

push @failures, $configuration->{'fail'};

}

}

#create new error message

my $status = CLCK::Result::Message->new($node);

$status->description('Check the appearance of fail words');

$status->value( 'found ' . $fail . " in line: \n\t" . $line );

$status->fail();

$status->severity(3);

...

push @status, $status;

...

Return

配置XML文件：

<cluster>

...

<test>

...

<dmesg>

<need>SATA</need>

<fail>fail</fail>

</dmesg>

...

</test>

</cluster>

XML配置文件必须由Intel Cluster CheckerXML validation Schema验证。因此用户需修改schema(<installation-path>/clck.xsd)。

<!-- test module name tags -->

<xsd:simpleType name="testModuleName">

<xsd:restriction base="xsd:token">

<xsd:pattern value="1GiB\_memory| … |X11\_libs|dmesg" />

<xsd:whiteSpace value="collapse" />

</xsd:restriction>

</xsd:simpleType>

…

<!-- dmesg test module -->

<xsd:element name="dmesg" minOccurs="0" >

<xsd:complexType>

<xsd:sequence>

<xsd:element name="fail" type="nonEmptyString" maxOccurs="unbounded"

minOccurs="0" />

<xsd:element name="need" type="nonEmptyString" maxOccurs="unbounded"

minOccurs="0" />

</xsd:sequence>

</xsd:complexType>

</xsd:element>

…

运行示例：

cluster-check config.xml –-include dmesg//命令行方式

<cluster> //配置文件方式

...

<include\_module>dmesg</include\_module>

...

</cluster>

cluster-check config.xml