如何做 Linpack 测试及性能优化

曹振南

czn@ncic.ac.cn，caozn@dawning.com.cn

2004 年 8 月

本文主要说明如何完成 HPL 测试，并介绍了一些简单的性能优化方法。

一、Linpack 简介

Linpack 是国际上最流行的用于测试高性能计算机系统浮点性能的 benchmark。通过对高性能计算机采用高斯消元法求解一元 N 次稠密线性代数方程组的测试，评价高性能计算机的浮点性能。

Linpack 测试包括三类，Linpack100、Linpack1000 和 HPL。Linpack100 求解规模为 100 阶的稠密线性代数方程组，它只允许采用编译优化选项进行优化，不得更改代码，甚至代码中的注释也不得修改。Linpack1000 要求求解 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。

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。

二、计算机计算峰值简介

衡量计算机性能的一个重要指标就是计算峰值或者浮点计算峰值，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。

理论浮点峰值＝CPU 主频×CPU 每个时钟周期执行浮点运算的次数×系统中 CPU 数

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，下表是各种 CPU 的每个时钟周期执行浮点运算的次数。

CPU	Flops/Cycle	CPU	Flops/Cycle	CPU	Flops/Cycle
IBM Power4	4	Ultra SPARC	2	Opteron	2
PA-RISC	4	SGI MIPS	2	Xeon	2
Alpha	2	Itanium	4	Pentium	1

实测浮点峰值是指 Linpack 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。

在实际程序运行中，几乎不可能达到实测浮点峰值，更不用说理论浮点峰值了。这两个值只是作为衡量机器性能的一个指标。

二、Linpack 安装与测试

1. Linpack 安装条件：

在安装 HPL 之前，系统中必须已经安装了编译器、并行环境 MPI 以及基本线性代数子方程(BLAS)或矢量图形信号处理库(VSIPL)两者之一。

编译器必须支持 C 语言和 Fortran77 语言。并行环境 MPI 一般采用 MPICH，当然也可以是其它版本的 MPI，如 LAM－MPI。HPL 运行需要 BLAS 库或者 VSIPL 库，且库的性能对最终测得的 Linpack 性能有密切的关系。常用的 BLAS 库有 GOTO、Atlas、ACML、ESSL、MKL 等，我的测试经验是 GOTO 库性能最优。

2. 安装与编译：

第一步，从 www.netlib.org/benchmark/hpl 网站上下载 HPL 包 hpl.tar.gz 并解包，目前 HPL 的最新版本为 hpl 1.0a。

第二步，编写 Make 文件。从 hpl/setup 目录下选择合适的 Make.<arch>文件 copy 到 hpl/目录下，如：Make.Linux_PII_FBLAS 文件代表 Linux 操作系统、PII 平台、采用 FBLAS 库；Make.Linux_PII_CBLAS_gm 文件代表 Linux 操作系统、PII 平台、采用 CBLAS 库且 MPI 为 GM。HPL 所列都是一些比较老的平台，只要找相近平台的文件然后加以修改即可。修改的内容根据实际环境的要求，在 Make 文件中也作了详细的说明。主要修改的变量有：

ARCH： 必须与文件名 Make.<arch>中的<arch>一致

TOPdir：指明 hpl 程序所在的目录

MPdir： MPI 所在的目录

MPlib： MPI 库文件

LAdir： BLAS 库或 VSIPL 库所在的目录

LAinc、LAlib：BLAS 库或 VSIPL 库头文件、库文件

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_DETAILED_TIMING ”为打印每一步所需的时间，缺省不打印

“ -DHPL_COPY_L ”为在 L 广播之前拷贝 L，缺省不拷贝（这一选项对性能影响不是很大）

CC： C 语言编译器

CCFLAGS：C 编译选项

LINKER：Fortran 77 编译器

LINKFLAGS：Fortran 77 编译选项（Fortran 77 语言只有在采用 Fortran 库是才需要）

第三步，编译。在 hpl/目录下执行 make arch=<arch>，<arch>即为 Make.<arch>文件的后缀，生成可执行文件 xhpl(在 hpl/<arch>/bin 目录下)

3. 运行：

运行 hpl 之前，需要修改配置文件 hpl.dat(在 hpl/<arch>/bin 目录下)，次配置文件每一项代表的意思在文档第三部分说明。

HPL 的运行方式和 MPI 密切相关，不同的 MPI 在运行方面有一定的差别。对于 MPICH 来说主要有两种运行方法。

1) 在 hpl/<arch>/bin 目录下执行：mpirun －np <N> xhpl。这种运行方式读取$(MPICH 安装目录)/share/machines.LINUX 配置文件

2) 在 hpl/<arch>/bin 目录下执行：mpirun －p4pg <p4file> xhpl。这种运行方式需要自己编写配置文件<p4file>，以指定每个进程在哪个节点上运行

MPICH 要求至少有一个 MPI 进程在递交任务的节点上运行，但 GM(MPI for Myrinet)、Infi－MPI(MPI for Infiniband)、ScaMPI(MPI for SCI)、BCL 等 MPI 来说，没有这个要求。LAM－MPI 我没怎么用过，所以不清楚其是否由此要求。

对于 GM 来说，可以采用 mpirun －machinefile <machinefile> -np <N> xhpl。这也是很多 MPI 所支持的一种运行方式，这种运行方式也需要自己编写<machinefile>以指定每个进程在哪个节点上运行

测试结果输出到指定文件中(在配置文件 hpl.dat 中定义)，缺省文件名为 HPL.out。

4. 查看结果

HPL 允许一`次顺序做多个不同配置测试，所以结果输出文件（缺省文件名为 HPL.out）可能同时有多项测试结果。

在文件的第一部分为配置文件 hpl.dat 的配置。在下面的部分

使用基准测试一般需要和收集的信息包括：

R：它是系统的最大的理论峰值性能，按 GFLOPS 表示。如 10 个 Pentium III CPU 的 Rpeak 值。

N：给出有最高 GFLOPS 值的矩阵规模或问题规模。正如拇指规则，对于最好的性能，此数一般不高于总内存的 80%。

Rmax：在 Nmax 规定的问题规模下，达到的最大 GFLOPS。

NB：对于数据分配和计算粒度，HPL 使用的块尺度 NB。小心选择 NB 尺度。从数据分配的角度看，最小的 NB 应是理想的；但太小的 NB 值也可以限制计算性能。虽然最好值取决于系统的计算/通信性能比，但有代表性的良好块规模是 32 到 256 个间隔。

```
================================================================================
T/V                N    NB     P     Q               Time             Gflops
--------------------------------------------------------------------------------
WC23C2C4      728480   232    32    80           31972.21          8.061e+03
--------------------------------------------------------------------------------
||Ax-b||_oo / ( eps * ||A||_1  * N        ) =        0.0028792 ...... PASSED
||Ax-b||_oo / ( eps * ||A||_1  * ||x||_1  ) =        0.0015927 ...... PASSED
||Ax-b||_oo / ( eps * ||A||_oo * ||x||_oo ) =        0.0002556 ...... PASSED
================================================================================
```

上面是我们在曙光 4000A Linpack 测试的最终结果。测试耗时 31972.21 秒＝8 小时 52 分 52 秒，实测浮点峰值为 8061Gflops＝8.061 万亿次/秒。

三、性能调优初步

作性能优化涉及的面很多，也很复杂，而且永无止境。对于不同的应用程序，优化的方法会有一些区别。我这里只阐述 Linpack 测试中一些性能优化方法，对于大型机群系统的 Linpack 测试可参见我写的论文《大规模 Linux 机群系统的 Linpack 测试研究》。这些优化方法不仅在 Linpack 测试有用，也可作为其它应用程序性能优化的参考。希望对大家有一些参考价值。

注：我一般采用的系统为基于 Opteron 和 Xeon 的两路或四路 SMP 机群系统，所以下面给出的一些经验值主要是我在上述系统中的一些测试经验，对于其它体系结构的 HPC 系统不一定适用，如 PVP、大型 SMP、NUMA 等等。

1．HPL.dat

以下是目前 HPL 最新版本 HPL 1.0a 的配置文件 hpl.dat。与 HPL 1.0 的配置文件相比，新的配置文件多了一个选项——第 9 行，处理器阵列的排列方式，是按行排列还是按列排列。在 1.0 中，其缺省为按列排列。

第1行	HPLinpack benchmark input file	
第2行	Innovative Computing Laboratory, University of Tennessee	
第3行	HPL.out	output file name (if any)
第4行	6	device out (6=stdout,7=stderr,file)
第5行	4	# of problems sizes (N)
第6行	29 30 34 35	Ns
第7行	4	# of NBs
第8行	1 2 3 4	NBs
第9行	1	PMAP process mapping (0=Row-,1=Column-major)
第10行	3	# of process grids (P x Q)
第11行	2 1 4	Ps
第12行	2 4 1	Qs
第13行	16.0	threshold
第14行	3	# of panel fact
第15行	0 1 2	PFACTs (0=left, 1=Crout, 2=Right)
第16行	2	# of recursive stopping criterium
第17行	2 4	NBMINs (>= 1)
第18行	1	# of panels in recursion
第19行	2	NDIVs
第20行	3	# of recursive panel fact.
第21行	0 1 2	RFACTs (0=left, 1=Crout, 2=Right)
第22行	1	# of broadcast
第23行	0	BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
第24行	1	# of lookahead depth
第25行	0	DEPTHs (>=0)
第26行	0	SWAP (0=bin-exch,1=long,2=mix)
第27行	32	swapping threshold
第28行	0	L1 in (0=transposed,1=no-transposed) form
第29行	0	U in (0=transposed,1=no-transposed) form
第30行	0	Equilibration (0=no,1=yes)
第31行	8	memory alignment in double (> 0)

下面逐个简要说明每个参数的含义，及一般配置。

1) 第 1、2 行为注释说明行，不需要作修改

2） 第 3、4 行说明输出结果文件的形式

“ device out ”为“ 6 ”时，测试结果输出至标准输出（stdout）

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

“ device out ”为其它值时，测试结果输出至第三行所指定的文件中

3） 第 5、6 行说明求解矩阵的大小 N

矩阵的规模 N 越大，有效计算所占的比例也越大，系统浮点处理性能也就越高；但与此同时，矩阵规模 N 的增加会导致内存消耗量的增加，一旦系统实际内存空间不足，使用缓存，性能会大幅度降低。因此，对于一般系统而言，要尽量增大矩阵规模 N 的同时，又要保证不使用系统缓存。

考虑到操作系统本身需要占用一定的内存，除了矩阵 A（N×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（具体占用的大小视不同的 MPI 而定）。一般来说，矩阵 A 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。

4） 第 7、8 行说明求解矩阵分块的大小 NB

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

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 的倍数等等。我在这里给出一些我的测试经验值，供大家参考。

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

根据我的测试经验，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是选择三个左右性能不错的 NB，在通过大规模测试检验这些选择。

5） 第 9 行是 HPL 1.0a 的新增项，是选择处理器阵列是按列的排列方式还是按行的排列方式。在 HPL 1.0 中，其缺省方式就是按列的排列方式。

按 HPL 文档中介绍，按列的排列方式适用于节点数较多、每个节点内 CPU 数较少的瘦系统；而按行的排列方式适用于节点数较少、每个节点内 CPU 数较多的胖系统。我只在机群系统上进行过测试，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。

6） 第 10～12 行说明二维处理器网格（P×Q）。二维处理器网格（P×Q）的要遵循以下几个要求：

- 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。这是一个测试经验值。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。
- $P=2^n$，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。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。

7) 第 13 行说明阈值。我读了 HPL 程序，这个值就是在做完线性方程组的求解以后，检测求解结果是否正确。若误差在这个值以内就是正确，否则错误。一般而言，若是求解错误，其误差非常大；若正确则很小。所以没有必要修改此值。

8) 第 14～21 行指明 L 分解的方式

在消元过程中，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 就采用了这三种。对这三种算法的具体描述可参考相关 LU 分解的资料，也可参加 HPL 的源代码，我在这里不过进一步的说明。

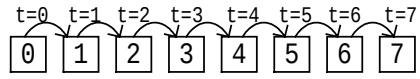
HPL 中，L 分解采用递归的方法，其伪代码如下：

```
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）
}
```

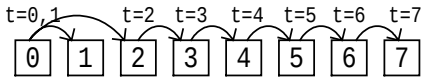
据我的测试经验，NDIVs 选择 2 比较理想，NBMINs 4 或 8 都不错。而对于 RFACTs 和 PFACTs，好像对性能的不大（在 LU 消元算法中，说 Crout 算法的性能不错）。我们对这两个参数作了大量的测试，没有发现什么规律，可能它是由于它对性能的影响较小，使得这个性能的微小差别由于机器的随机性而无法区分。

9) 第 22、23 行说明 L 的横向广播方式，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。

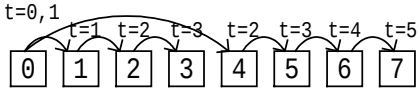
前四种算法如图所示，分别采用单环/双环、第一列处理器不优先/优先。



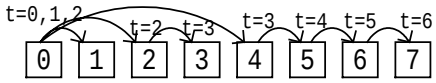
0)Increasing-ring:
单环,不优先



1)Increasing-ring(M):
单环,优先



2)Increasing-2-ring:
双环,不优先



3)Increasing-2-ring(M):
双环,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。

根据我的测试经验，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。

10） 第 24、25 行说明横向通信的通信深度。

这部分由于时间关系代码读的不是非常明白，大体的意思是这样的。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。

根据我的测试经验，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。

11） 第 26、27 行说明 U 的广播算法。

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 $\log_2 P \times$（Latency＋NB×LocQ（N）/Bandwith），适用于通信量较小的情况；Long 法的通信开销为（$\log_2 P$＋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。

12） 第 28、29 行分别说明 L 和 U 的数据存放格式。

大家知道，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。我没有对这两个选项进行性能测试，而选择按列存放。因为感觉按列存放是，性能会好一些。

13） 第 30 行主要是在回代中用到。由于回代在整个求解过程中占的时间比例非常小，所以由于时间关系，我对这部分程序读的很少，其具体的含义不是很清楚。我一般都用其缺省值。

14） 第 31 行的值主要为内存地址对齐而设置，主要是在内存分配中作地址对齐而用。

2．其它性能优化

除了对 HPL 配置文件进行调整外，还有其它很多的优化方法。

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。

2） 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。

下面是在 XEON 平台上的测试结果，可供参考。

测 试 环 境		
硬件环境	节点数	16 个
	处理器	2 路 Intel Xeon 2.8 GHz / 512K L2 Cache
	内存	2G
	网络	千兆以太网(1000Base－T)
	交换机	3COM 17701 千兆以太网交换机
软件环境	操作系统	Redhat Linux 8.0
	编译器	GNU C/C++ 3.2 GNU F77 3.2 Intel C/C++ Compiler v7.1 for Linux Intel Fortran Compiler v7.1 for Linux PGI 5.0
	并行环境	MPICH-1.2.5.2 (采用 gcc/g77 编译，ch_p4 的通讯方式)
	BLAS 库	ATLAS 3.4.1 MKL 5.2 GOTO 0.9

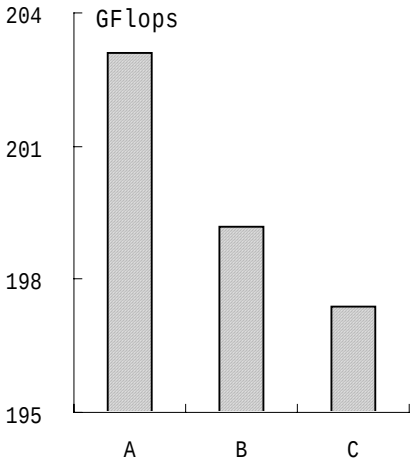
节点/进程	规模/分块	ATLAS	MKL	GOTO
1 节点 1 进程	15000/400	3.545		4.479
	15000/384	3.514		4.462
	15000/192	3.431		4.486
1 节点 2 进程	15000/400	6.111	6.228	7.977
	15000/384	6.036	6.379	7.911
	15000/192	5.756	5.795	8.029

3） 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。

	A	B	C
Process0	node0	node0	node0
Process1	node1	node0	node0
Process2	node2	node1	node0
Process3	node3	node1	node0
Process4	node4	node2	node1
Process5	node5	node2	node1
Process6	node6	node3	node1
Process7	node7	node3	node1
Process8	node0	node4	node2
Process9	node1	node4	node2
… …	… …	… …	… …
Process30	node6	node7	node7
Process31	node7	node7	node7

A



B

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。

4） 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，几乎每一种优化都

我这里只是说最简单的一种方法，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。

5） 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。77

6） 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。

7） 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。

四、参考文献

[1] http://www.netlib.org/benchmark/hpl

[2] 曹振南，冯圣中，冯高峰．曙光 4000A Linpack 测试技术报告．中科院计算所智能中心技术报告．2004．

[3] 曹振南，冯圣中，冯高峰．大规模 Linux 机群系统的 Linpack 测试研究．第八届全国并行计算学术交流会(NPCS)．2004.07．