# PLDA使用手册

## 一、介绍

### 1. LDA

隐含狄利克雷分布简称LDA (Latent Dirichlet allocation)，是一种用于离散数据集合如文本语料库的生成概率模型。它是 文档-主题-词（document-topic-word）三级分层的结构：LDA将文档集合中的每篇文档看作是不同主题构成的，主题是按照一定概率分布的；而每个主题中的词也有其概率分布。

同时LDA也是一种无监督学习算法，在训练时不需要手工标注，仅仅指定主题的数量，他就可以自动发现文档的主题。训练的结果是以概率分布的形式给出的，即一篇文档不是确定的属于某个主题，而是对于每个主题有不同的概率。此外LDA的另一个优点则是，对于每一个主题均可找出一些词语来描述它。

LDA首先由David M Blei、吴恩达和Michael I Jordan于2003年提出，目前在文本挖掘领域包括文本主题识别、文本分类以及文本相似度计算方面都有应用。LDA是一种非监督机器学习技术，能够识别大规模文档集或语料库中潜在的主题信息，同时还能够预测推断一篇新的文档和哪些主题相关。LDA 模型现在已经成为了主题建模中的一个标准。LDA 模型自从诞生之后有了蓬勃的扩展，特别是在社会网络和社会媒体研究领域最为常见。

LDA能够识别大规模文档集中的潜在主题信息，同时还能够预测推断（inference）一篇新文档和哪些主题（topic）相关。LDA采用的是词袋（bag of words）的方法, 即它认为一篇文档是由一组词构成的一个集合，词与词之间没有顺序以及先后的关系，只有使用的频率。一篇文档可以包含多个主题，文档中每一个词都由其中的一个主题生成。如果两个不同的词经常一起出现在文档中，那么LDA的训练算法倾向于把这两个词归于同一个主题。

一般来说LDA的实现包括训练算法和预测算法两个部分：训练算法是指基于已有的文档集，学习出LDA模型，LDA模型的效果和模型收敛情况有较大影响；预测算法是指利用已学习出来LDA模型去推断一篇新文档的主题分布。

LDA假设文档是这样产生的，要产生一篇文档，首先要生成该文档的主题分布，即一篇文档会包含哪些主题以及每个主题所占的比例；然后要生成这篇文档中的所有词，每个词的生成需要根据该文档中主题的概率分布随机选择一个主题，再根据该主题中词的概率分布随机生成一个词。基于这种假设，LDA根据现实的大量文档集，训练得到主题的概率分布和每个主题中词的概率分布。

由于LDA的训练需要成千上百次迭代，每次迭代需要遍历所有训练文档的所有词，并且不断更新其中的 文档-主题 矩阵（大小为 D × K，其中D为文档个数、K为主题个数）、主题-词 矩阵（大小为K × V，其中V为词数）, 非常耗时，所以训练算法并行需求很大。

### 2. PLDA软件

PLDA是由Google实现的MPI并行版本的LDA软件，它采用高度优化的并行Gibbs采样算法，可以用于LDA训练及预测判断。PLDA解决了存储和计算的瓶颈，并且对于长时间的分布式计算提供了错误恢复机制。它的源代码可以在其主页下载到，https://code.google.com/p/plda/。

但是PLDA在内存和存储、以及扩展性方面都遇到了瓶颈，为了解决这些问题，中科院网络中心在MPI并行版本LDA软件的基础上进行了进一步优化：一是修改了处理的数据结构，优化了内存的使用，减少内存使用量；二是引入了OpenMP多线程并行，将LDA增强为两级混合并行模式，提高了程序的可扩展性，使其能够在更大规模的机器上良好运行；三是增加了预处理，过滤掉出现频率低的词，从而在保证训练结果准确的前提下，减少了计算量。

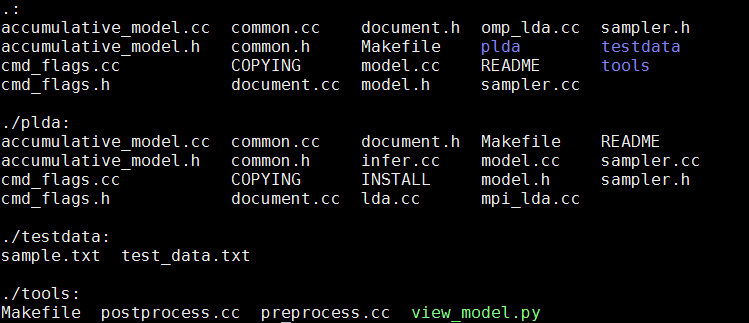
## 二、安装

### 1. 下载源代码

下载plda-omp.tar.gz，将它放置在想要的目录下，并解压缩

tar xvfz plda-omp.tar.gz  
cd plda-omp

运行ls列出所有的文件



源代码中包括以下的程序：

* omp\_lda：基于MPI以及OpenMP的混合并行PLDA训练程序

plda目录下存放是是原始的PLDA 3.1版本，其预测推断程序infer是需要的

* lda：单处理器版本的LDA训练程序
* mpi\_lda：基于MPI的并行PLDA训练程序
* infer：单处理器版本预测推断程序，使用训练程序得到的LDA模型对新文档的主题进行预测推断

testdate目录

* sample.txt：本文档中各节示例中使用的小数据集，用于说明数据格式
* test\_date.txt：测试用数据集文件

Tools目录

* preprocess：预处理程序，将训练数据集中的词替换为唯一编号，并生成字典文 件
* postprocess：后处理程序，将模型文件中的唯一编号对应回词
* view\_model.py：将训练程序生成的模型文件转换成可读文本的Python程序

### 2. 修改Makefile

编译PLDA源代码要求系统中安装有C++编译器以及MPI环境，C++编译器有GCC、Intel Composer以及PGI等不同选择，MPI环境也有MPICH、Intel MPI、Open MPI等等多种选择，这些编译器和MPI环境的安装和使用请参阅各自的使用手册。

根据使用的编译器和MPI环境的不同，需要修改Makefile中的三个参数才能保证编译正确进行：

* CC：C++编译器，可以是g++、icpc等
* MPICC：并行版本使用的MPI C++编译器，可以是mpicxx、mpiicpc等
* CFLAGS：编译器参数

例如，使用GCC进行编译、MPI环境为MPICH时，这三个参数可以为：

CC=g++  
MPICC=mpicxx  
CFLAGS=-O3 -Wall -Wno-sign-compare

而使用Intel Composer作为编译器、MPI环境为Intel MPI时，这三个参数就需要改为：

CC=icpc  
MPICC=mpiicpc  
CFLAGS=-O3 -ipo -xHost -Wall -Wno-sign-compare

### 3. 编译

修改好Makefile之后，只需要执行make命令就可以进行编译

make

编译完成之后会产生omp\_lda和infer的执行文件，以及plda目录下的lda、mpi\_lda和infer，tools目录下的preprocess和postprocess。

如果需要单独编译某一个版本或者单独编译infer，可以这样执行

omp\_lda：

make omp\_lda

lda：

make lda

mpi\_lda：

make mpi\_lda

infer：

make infer

tools/preprocess：

cd tools  
make pre

tools/postprocess：

cd tools  
make post

## 三、准备数据文件

### 1. 数据文件格式

LDA训练程序需要有文档集作为输入文件才能得到LDA模型，而原始的文本文档是不能够直接进行处理的，需要处理成LDA训练程序能够接受的数据文件。

数据文件是使用稀疏格式来存储的，每一个文档占用一行,每一行是这个文档中的所有词以及每个词在文档中的数量。看起来就像这样：

<词1> <词1的数量> <词2> <词2的数量> <词3> <词3的数量> ...

每个词都是任意的字符串，但是不包含空格、换行或其它特殊字符。

例如：假设有两个文档:第一个是“a is a character”；第二个是“b is a character after a”。那么数据文件看起来是这样的：

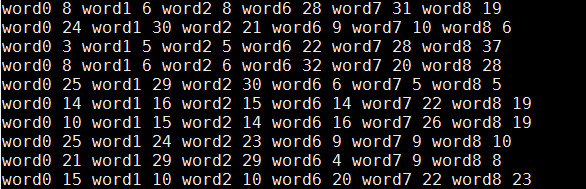
a 2 is 1 character 1  
a 2 is 1 b 1 character 1 after 1

当文档集很大时，输入的数据文件会非常大。

在源代码的压缩包里带有一份训练样例数据文件，test\_data.txt，存放在testdata/目录下，可以用来进行测试。在对实际的文档集进行训练时，需要使用命令行参数--training\_data\_file来指定数据文件的目录以及文件名。

### *示例*

下面是一个简单的测试文档集，在本手册中会以此为例介绍训练和预测推断等各个步骤的输出结果格式。



可以看出，这个文档集中一共包含十篇文档，每篇文档都是由六个词组成的，这六个词在每篇文档中的词频各不相同。

### 2. 预处理/后处理

混合并行版本的PLDA软件为了减少内存使用量以及加快计算速度，对于原始版本PLDA中处理的数据结构进行了一些改进，所以原来格式的数据文件不能直接进行计算，需要使用预处理程序将其转换为可以处理的新格式。并且在训练得到模型文件后，需要使用后处理程序将其转换回预测推断程序infer可处理掉格式。

文档被分布到各个节点，每个节点都有一份 主题-词 矩阵的副本，每次迭代都会对该矩阵进行更新。文档-主题 矩阵则是虽着进程数增加，而在每个进程所占的空间减少，但是 主题-词 矩阵在每个进程占用存储空间固定。为了降低相关数据结构的存储负载，对输入和输出文件进行了预处理和后处理操作，具体如下：

1. 去除词频过低的词。对出现频率过低的词并不关注，去除掉这些词可以降低存储空间使用以及减少计算量，比如从文档集合中去除出现次数少于10次的词。
2. 在输入文件中将词替换为唯一编号项。由以前<词（字符串），文档中词出现的次数（整数）>变为 <词唯一编号（整数），文档中词出现的次数（整数）>，在程序处理过程中用词的唯一编号来代替词（字符串）以降低程序运行中的存储空间。同时生成字典文件vocab\_file存储唯一编号与词的对应关系，供后处理程序使用。
3. 在进行预测后，执行后处理，将生成的模型文件中单词唯一编号对应回实际的词，供预测推断程序infer使用。

预处理程序preprocess的运行方式如下：

./tools/preprocess --input testdata/sample.txt --output testdata/sample\_index.txt --vocab\_file /tmp/sample\_vocab.txt --threshold 160

它一共有四个参数，其含义分别如下：

* input：输入的数据文件。
* output：输出的数据文件，使用唯一编号替换了词，并且去除了出现次数（词频）小于threshold的词。
* vocab\_file：字典文件，存储 词-唯一编号 对应关系。
* threshold：进行去除低频词的阈值，如果不输入这个参数，取默认值0，即不进行过滤。

后处理程序postprocess的运行方式如下：

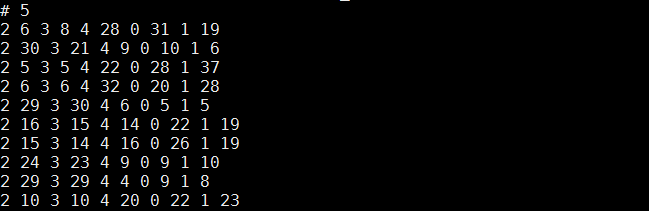
./tools/postprocess --vocab\_file /tmp/sample\_vocab.txt --input\_model /tmp/index\_model.txt --output\_model /tmp/lda\_model.txt

它一共有三个参数，其含义分别如下：

* vocab\_file：字典文件，存储 词-唯一编号 对应关系。
* input\_model：输入的模型文件。
* output\_model：输出的模型文件，将生成的模型文件中单词唯一编号对应回实际的词，供预测推断程序infer使用。

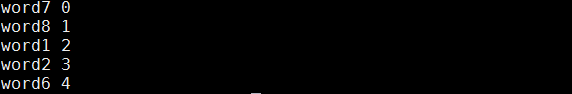
### *示例*

对于测试文档集sample.txt取threshold=160进行预处理得到的结果如下：



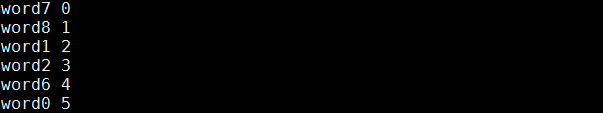
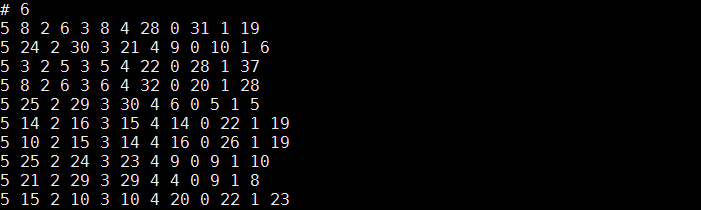
可以看出它比原始文件明显短了很多，处理过的数据文件在第一行增加了注释，这个数值代表经过处理后在整个数据文件中出现的词的个数，这个例子中就一共出现了5个词，它们的词频都大于160。

生成的字典文件如下：



这个字典文件分为两列，第一列是词，第二列是词对应的唯一编号，对比原始数据文件可以看出word0因词频低于160被过滤掉了。

因为实际word0与其它词的词频比较接近，在后面的示例中依然采用不过滤的数据进行处理，即数据文件和字典文件分别为：



后处理的示例在后面训练程序示例中一并给出。

## 四、训练

准备好数据文件之后，就可以使用LDA训练程序进行训练，下面以使用源代码自带样例数据文件为例说明如何运行LDA训练程序，而命令行参数的详细说明在下一节。

输入的数据文件为testdata/test\_data.txt，需要生成2个主题，输出的模型存放在/tmp/lda\_model.txt中，训练一共进行150次Gibbs迭代，alpha和beta两个参数的取值为0.1和0.01。

* 单处理器版本

单处理器版本有一个特别的参数，burn\_in\_iterations，它的含义在下一节详细介绍，在本例子中进行100次迭代。

./plda/lda --num\_topics 2 --alpha 0.1 --beta 0.01 --training\_data\_file testdata/sample.txt --model\_file /tmp/lda\_model.txt --burn\_in\_iterations 100 --total\_iterations 150

* PLDA MPI并行版本

例子中使用8个MPI进程进行并行计算。

mpirun -np 8 ./plda/mpi\_lda --num\_topics 2 --alpha 0.1 --beta 0.01 --training\_data\_file testdata/sample.txt --model\_file /tmp/lda\_model.txt --total\_iterations 150

* PLDA混合并行版本

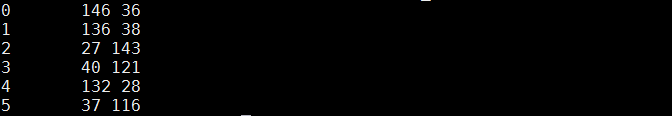
PLDA混合并行版本与MPI并行版本的参数都相同，但是执行时需要通过环境变量OMP\_NUM\_THREADS指定每进程的并行数，例子中使用2个MPI进程，每个MPI进程使用4个线程。

Export OMP\_NUM\_THREADS=4  
mpirun -np 2 ./omp\_lda --num\_topics 2 --alpha 0.1 --beta 0.01 --training\_data\_file testdata/sample\_index.txt --model\_file /tmp/index\_model.txt --total\_iterations 150

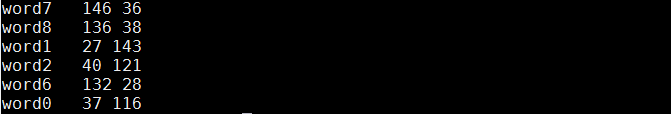
训练完成后，会看到产生一个/tmp/index\_model.txt文件。这个文件存储了训练结果。每一行是一个词的主题分布。第一个元素是词的唯一编号，然后是它在每个主题中的出现次数。需要使用后处理程序将唯一编号对应回词，然后可以使用view\_model.py文件将模型转换成可读的文本。

### *示例*

对于示例测试文档进行训练，进行两个主题的聚类，得到如下的模型文件：



经过后处理后，将唯一编号替换回词后可以更清楚的看到词的概率分布：



学习的结果是两个主题中六个词的概率分布，即：

在主题一中，word0的概率是37、word1的概率是27、word2的概率是40、word6的概率是132、word7的概率是146、word8的概率是136；

在主题二中，word0的概率是116、word1的概率是143、word2的概率是121、word6的概率是28、word7的概率是36、word8的概率是38。

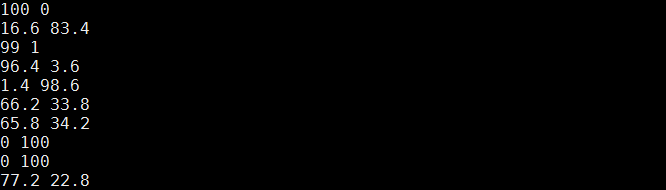
## 五、预测推断

在训练得到LDA模型lda\_model.txt后，infer程序就可以根据它对于新的未见文档预测推断其主题

./infer --alpha 0.1 --beta 0.01 --inference\_data\_file testdata/sample.txt --inference\_result\_file /tmp/inference\_result.txt --model\_file /tmp/lda\_model.txt --total\_iterations 15 --burn\_in\_iterations 10

### *示例*

利用已经得到的模型再对测试文档集进行预测推断：



推测结果一共十行，代表文档集中的十篇文档；每一行有两个数字，代表两个主题的概率分布。以文档一为例，它100%属于主题一，0%属于主题二。

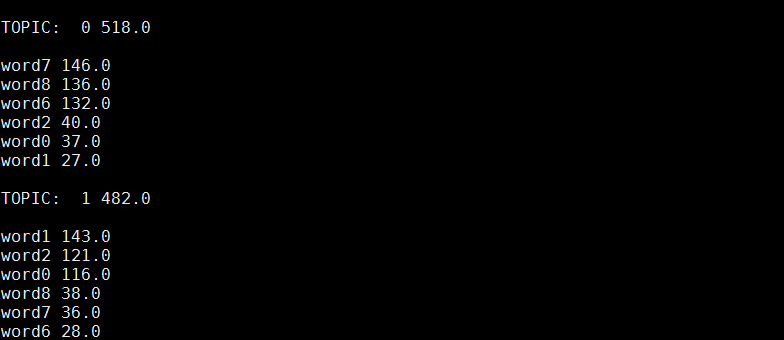
## 六、将模型文件转换为可读文本

LDA训练程序生成的模型文件可读性很差，Python程序view\_model.py可以将其转换为可读性更好的文本，它按照 主题-词 的方式将模型列出。

./tools/view\_model.py /tmp/lda\_model.txt /tmp/viewable\_model.txt

### *示例*

对于测试文档集训练生成的模型文件转换成可读文本：



可以看出它比原始模型文件更容易理解，列出了每个主题的总概率分布，主题中词的概率分布按照降序排列。

## 七、命令行参数

### 1. 训练时参数

* **num\_topics**：主题（topic）数量，由用户指定
* **alpha**：主题分布的生成参数，建议设为50/number\_of\_topics
* **beta**：词分布的生成参数，建议设为0.01
* **training\_data\_file**：训练使用的数据文件
* **model\_file**：训练模型的输出文件
* **total\_iterations**：Gibbs采样迭代次数
* **burn\_in\_iterations**：在burn\_in\_iterations次循迭代后，模型已经基本收敛。 然后计算最后(total\_iterations - burn\_in\_iterations)次迭代的平均值作为最终模型。**这个参数只在单处理器版本中起作用**。例如：将total\_iterations设为200，会发现在170次迭代后，模型已经基本收敛。这样可以将burn\_in\_iterations设为170，这样最终模型就是最后30次迭代的平均值。

### 2. 预测推断时参数

* **alpha**和**beta**：应该设为和训练时同样的值
* **inference\_data\_file**：进行预测判断的未见文件
* **inference\_result\_file**：对未见文件进行预测判断的输出结果
* **model\_file**：训练得到的模型文件
* **total\_iterations**：对于未见文档进行Gibbs采样迭代的次数，以决定它的词主题。这个次数不需要像训练时那么大，通常几十次就可以了。
* **burn\_in\_iterations**：对于未见文档，计算最后(total\_iterations - burn\_in\_iterations)次迭代的 文档-主题 分布的平均值作为最终的 文档-主题 分布。

## 八、测试

### 1. 正确性测试

为了验证PLDA在不同软件环境下编译和运行的正确性，选取了以下6台测试机进行了验证：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 测试机一 | 测试机二 | 测试机三 |
| 处理器 | 双路英特尔至强处理器X5680，主频3.33GHz，6核12线程 | 双路英特尔至强处理器E5-2670，主频2.6GHz，8核16线程 | 双路英特尔至强处理器E5-2699 V3，主频2.3GHz，18核36线程 |
| 微架构 | Westmere | Sandy Bridge | Haswell |
| OS | CentOS 6.3 | RHEL 6.3 + CentOS 6.3 | RHEL 6.5 |
| 编译器 | GCC 4.4.6 + ICC 14.0.3 | GCC 4.4.6 + ICC 14.0.3 + ICC 15.0.2 | GCC 4.4.7 + ICC 14.0.3 + ICC 15.0.2 |
| MPI | Intel MPI 5.0 | Intel MPI 4.1.3 + Intel MPI 5.0.3 | Intel MPI 4.1.3 + Intel MPI 5.0.3 |
|  | 测试机四 | 测试机五 | 测试机六 |
| 处理器 | 四路英特尔至强处理器E7-4870，主频2.4GHz，10核20线程 | 四路英特尔至强处理器E7-8890 V3，主频2.5GHz，18核36线程 | 英特尔至强融核协处理器7120A），主频1.238GHz，61核244线程 |
| 微架构 | Westmere | Haswell | Knights Corner |
| OS | RHEL 6.0 | CentOS 7.0 | RHEL 6.5 + MPSS 3.5 |
| 编译器 | GCC 4.4.4 + ICC 12.1.2 | GCC 4.8.3 + ICC 15.0.0 | ICC 15.0.2 |
| MPI |  | Intel MPI 5.0.1 | Intel MPI 5.0.3 |

在以上6台测试机的这些软硬件环境基本覆盖了自2010年以来主流的HPC软硬件应用环境。

在所有这些环境中，均对三个版本的LDA训练程序进行了测试，其结果完全正确，说明其对于软硬件环境具有广泛的适应性。

值得一提的是在测试机六上，即英特尔至强融核协处理器上，源代码不需要进行任何的修改，仅需将Makefile中CFLAGS修改为：

CFLAGS=-O3 -ipo -mmic -Wall -Wno-sign-compare

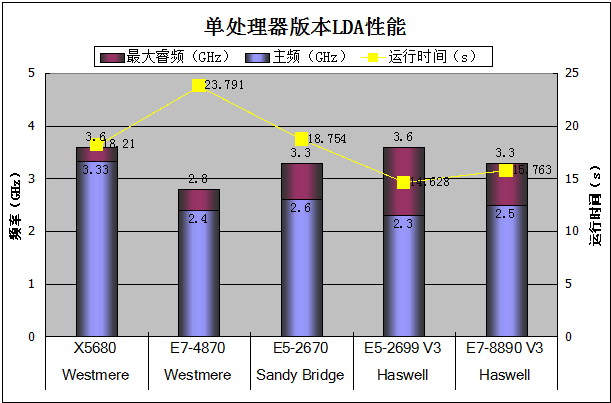
仅仅修改一个参数，经过编译后，程序就可以在协处理器上以Native的方式直接运行，与在CPU上的运行方式完全相同。

### 性能测试

#### 2.1 单核性能比较

在进行单核测试时，选择使用了单处理器版本的LDA程序，目的是检查处理器的微架构、主频以及编译器的选择对于LDA程序性能的影响。测试环境采用了正确性测试同样的测试机一至测试机五，一共五台机器。编译器均采用英特尔C++编译器。测试成绩如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 处理器型号 | 微架构 | 主频（GHz） | 最大睿频（GHz） | 运行时间（s） |
| X5680 | Westmere | 3.33 | 3.6 | 18.210 |
| E5-2670 | Sandy Bridge | 2.6 | 3.3 | 18.754 |
| E5-2699 V3 | Haswell | 2.3 | 3.6 | 14.628 |
| E7-4870 | Westmere | 2.4 | 2.8 | 23.791 |
| E7-8890 V3 | Haswell | 2.5 | 3.3 | 15.763 |



从上面的图中可以很明显的看到，在同一种微架构中，LDA程序的性能与频率之间几乎是线性关系，因为在这三代英特尔处理器中均具有睿频技术（也就是在功耗、温度等条件允许的前提下降CPU核心的运行频率提高到标准主频之上，最高可以达到最大睿频），所以在比较性能与频率的关系时使用的是最大睿频值。

而对于不同微架构的处理器，可以很明显地看到越新的微架构性能越好，Haswell快于Sandy Bridge，Sandy Bridge又稍快于Westmere。

在E5-2699 V3和E7-8890 V3这两款处理器，也同时比较了采用GCC编译器和ICC编译器的性能差别，因为两台机器上安装的GCC编译器版本不同，所以其运行时间无法互相比较，而ICC的版本相同，其运行时间可以互相比较。

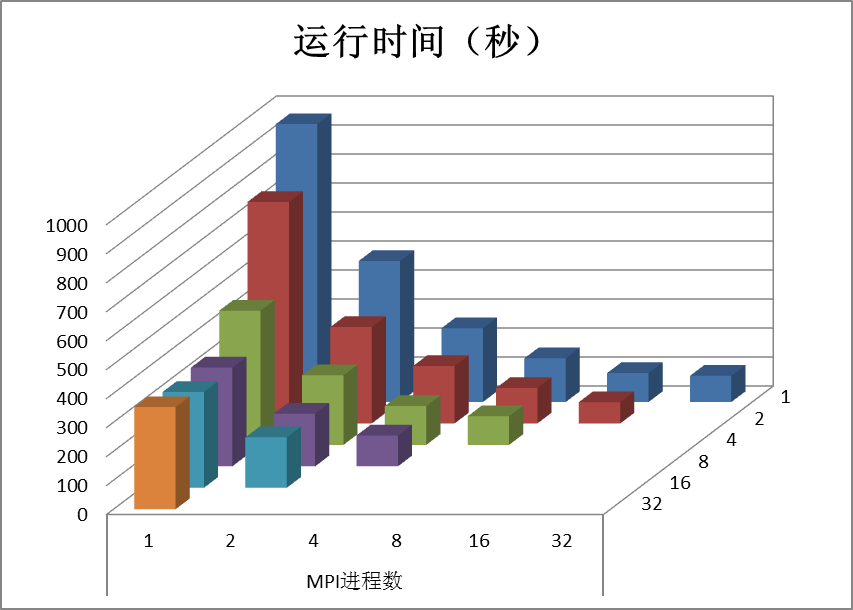
|  |  |  |  |
| --- | --- | --- | --- |
| 处理器型号 | GCC运行时间（s） | ICC运行时间（s） | 性能提升 |
| E5-2699 V3 | 19.547 | 14.628 | +33.63% |
| E7-8890 V3 | 18.587 | 15.763 | +17.92% |

可以看到Intel编译器比起GCC编译器有很大的性能优势，对比不同版本提升幅度在>10%到>30%不等。

#### 2.2 多核性能

多核测试采用混合并行版本的PLDA程序，目的是检验PLDA程序的扩展性。在五台服务器组成的小集群上进行测试，每台服务器均配置：两块Intel至强处理器E5-2699 v3，64GB DDR4-2133内存，Mellanox QDR IB网卡；存储使用Panasas网络存储，通过10GbE以太网连接。对于100MB大小的网页数据进行100步300个主题的训练，其中包括745369个文档，去threshold=10进行过滤后，字典中一共有37790个不同的词。各种不同组合的MPI进程数×OpenMP线程数的运行时间如下表所示，单位为秒：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| MPI进程数 | OpenMP线程数 | | | | | |
| 1 | 2 | 4 | 8 | 16 | 32 |
| 1 | 958.719 | 764.284 | 463.113 | 340.663 | 330.513 | 352.200 |
| 2 | 486.498 | 332.531 | 240.061 | 181.66 | 173.937 |  |
| 4 | 254.824 | 198.355 | 133.853 | 106.476 |  |  |
| 8 | 150.698 | 122.101 | 98.888 |  |  |  |
| 16 | 99.859 | 73.509 |  |  |  |  |
| 32 | 90.377 |  |  |  |  |  |



很明显，对于在32个核心上对于同样的数据集进行训练，采用的MPI进程数量与OpenMP线程数量不同，程序的运行时间差别很大，32进程1线程执行时运行时间为90.377秒，而1进程32线程执行时运行时间为352.200秒，差别达到3.89倍。进程与线程的扩展性差别明显，要取得最好的结果，要对于不同的进程数和线程数的组合进行测试，这里的测试结果是采用16进程2线程能达到最快的性能。同时也说明采用MPI/OpenMP两级混合并行能够提升性能。

#### 与前代处理器性能比较

对于如下的大测试集在5个Intel Xeon E5-2699 v3节点进行测试，并与之前在32个Intel Xeon E5450节点上的测试结果进行比较:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 迭代步数 | 主题 | 文档数 | 字典词数 | α | β |
| 1200 | 300 | 1.5亿 | 1892263 | 0.1 | 0.01 |

2007年发布的E5450采用Harpertown架构，比2014年发布的E5-2699 v3的Haswell架构要早五代。E5450使用了256个3.0GHz主频的核心，E5-2699 v3进行了两组测试，分别使用了144个和180个2.3GHz主频的核心。测试的结果如下：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 256个E5450核心 | 144个E5-2699 v3核心 | 180个E5-2699 v3核心 |
| 运行时间 | 25小时6分钟 | 21小时27分种1秒 | 16小时47分钟18秒 |

可以看出，一颗E5-2699 v3的核心要等于两颗以上E5450核心；而从节点数量上对比，一个E5-2699 v3节点要等于9至10个E5450节点。从成本的角度，新型号处理器具有很大的优势。

## 参考文献

1. Blei, D.M., Ng, A.Y., Jordan, M.I.: *Latent Dirichlet allocation*, Journal of Machine Learning Research (2003)
2. Edwin Chen, *Introduction to Latent Dirichlet Allocation*, http://blog.echen.me/2011/08/22/introduction-to-latent-dirichlet-allocation/, 22 August 2011
3. *Latent Dirichlet allocation*, https://en.wikipedia.org/wiki/Latent\_Dirichlet\_allocation
4. Yi Wang, Hongjie Bai, Matt Stanton, Wen-Yen Chen, and Edward Y. Chang, *PLDA: Parallel Latent Dirichlet Allocation for Large-scale Applications*, AAIM 2009
5. Zhiyuan Liu, Yuzhou Zhang, Edward Y. Chang, Maosong Sun, *PLDA+: Parallel Latent Dirichlet Allocation with Data Placement and Pipeline Processing*, ACM Transactions on Intelligent Systems and Technology, special issue on Large Scale Machine Learning (2011)
6. *A Quick Start Manual for plda*, https://code.google.com/p/plda/wiki/PLDAQuickStart
7. *A Manual on How to Train a Model Parallelly*, https://code.google.com/p/plda/wiki/PLDAManual