并行程序设计实验

实验 3 文档分类向量化

联系方式：NJU-TJL@outlook.com

目录

实验 3 文档分类向量化

一、 实验设计

1. MPI 实现部分

MPI 程序的大体框架参考了讲义给的《MPI 与 OpenMP 并行程序设计：C 语言版》的第 9 章内容。通过阅读和理解，掌握了其中给出的管理者/工人模式的并行程序大致思路，而剩下的一些内容如（如读取文件目录、构造哈希等等），还需要自己完善。故此次实验先完成了 MPI 部分并行程序。下面对整体思路和一些具体实现细节进行阐述。

1）整体思路

原书已给出较为完整的设计思路、管理者/工人进程伪代码和 C 语言代码。故这里不再用过多篇幅赘述，仅仅说一下在阅读之后自己的一点理解。

首先，总体上的任务是读取指定目录下的文本文件及字典文件，然后统计每个文档中字典中的每个词的出现个数（即构成“文档向量”），并写入一个结果文件。

此任务中，任务的数目在编译时无法确定，而各个任务之间不需要通信，完成不同任务处理每个文档所花费的时间差异可能会较大，因为文档的大小可能有很大的不同。所以这里采用的是管理者/工人模式进行。

管理者负责：①读取文件目录，获取待处理的文件是哪些。②分配文档处理任务给工人进程，负责总体调度（比如什么时候终止）。③接收工人进程处理结果（即文档向量），并汇总输出到结果文件。

工人负责：①0 号工人负责读取字典，然后广播给所有工人。②从字典文件构造哈希表，便于后面的文档向量化处理。③完成管理者指定的文档处理任务，并传送给管理者处理结果（即文档向量）。

2）读取字典，构建哈希表

首先，0 号工人进程需要读取字典文件内容，并且广播给所有工人。实现中，0 号工人调用 readAll 函数将字典文件的内容一次性读入到 buffer 中，然后在广播给所有工人。之后，每个工人进程都调用 make_dict_Hash 函数，根据字典文件内容来构建 Hash 表。这种方式相比于另一种可能的实现（每个工人进程都各自读取字典文件内容），是更高效的。因为读取同一个文件，涉及文件 I/O 操作，而且可能还有占用问题，往往较慢。而进程间广播速度则会更快一些。

对于 Hash 表的构建，这里的实现如下。首先，这里构建了一个 Hash 结构，输入一个字符串（字典中的某个词），可以在较低（接近常数）的复杂度内返回一个整数，表示在字典中的下标，即这个词是字典中的第几个词。

对字符串的 Hash 函数有很多，这里采用了 P.J.Weinberger 提出的一个函数，输入一个字符串，Hash 结果为一个小于等于 0x3fff 的整数。而 Hash 表采用的是链地址法构建，即一个大小为 0x3fff+1 的数组。这样即可解决 Hash 地址冲突的问题（冲突则放到该地址的同一个链表上）。其中存储着链表头节点，每个链表上才是存储真正的 Hash 节点（即字符串的内容和对于的字典下标值）。

有了 Hash 表之后，文档向量化就变得简单：读入文档中的一个单词，查 Hash，获得单

词在字典中的下标，表示文档向量的数组对应下标的元素加 1 即可。

3）读取文件目录，识别文档

管理进程需要在用户给出的目录下识别出 n 个文本文档。这里实现了 get_names 函数来完成这一功能。即：在指定目录下，识别出所有普通文件，构建一个记录文件路径的数组，且返回文件个数。

这里主要用到了 Linux 下提供的 dirent.h 中的相关 API 函数来完成，用类似读取文件的方式来读取目录。首先，需要使用 opendir 函数打开目录，然后可以使用 readdir 对目录进行遍历，访问其中的文件、文件夹等内容。这里首先进行了一次遍历，得到文件总数。分配好数组空间后，第二次遍历记录下所有文档的路径，存在数组中，最后再对这个数组按照文件路径排个序。

2. OpenMP 实现部分

实现 OpenMP 程序，首先实现一个串行版本。而在 MPI 程序的实现基础上，实现串行程序则非常简单。一些模块的操作，比如读取字典文件、构造 Hash、读取目录识别文档、文档向量写入结果文件等，都以函数的形式封装在了 MyUtils.c 中，所以串行程序只需将这些过程组合起来即可。

1）串行程序思路

串行程序实现非常简单，相比 MPI 实现少了很多进程通信、任务分配等的额外操作。

主要处理步骤如下：

a) 读取字典文件；
b) 构建哈希表；
c) 读取目录，识别文档文件，数组中记录下所有文档的路径，以及文档个数 n；
d) 用一个 for 循环，依次对 n 个文档进行向量化；
e) 文档向量写入结果文件。

2）并行化思想

串行程序中，程序主要执行过程在上述过程的 d)，而这一步中，是一个 for 循环的过程，非常适合并行化。所以这里的并行化非常简单，只需要在 for 循环前加一行 "#pragma omp parallel for" 即可。

二、 实验结果

注：关于程序正确性的验证。

测试时使用了一些小规模的字典文件（比如只含有 a 开头的前 10 个单词）和几个小的文档（内容只有几个单词），OpenMP 和 MPI 程序的测试结果均为正确。此外，对于老师给定的样例（100 个文档），将 OpenMP 和 MPI 程序输出的结果文件使用 Linux 终端下的 diff 命令进行比对，发现完全一致，所以基本可以确定 OpenMP 和 MPI 所实现的文档向量化程序逻辑无误。

1. 运行环境

物理机 CPU：Intel i5-7200U （双核），内存 16GB
虚拟机：CPU 分配 2 个核，内存 4GB
虚拟机系统环境：Ubuntu 16.04（64 位版本）、gcc version 5.4.0 20160609、mpicc for MPICH version 3.3.2

2. MPI 实现运行结果

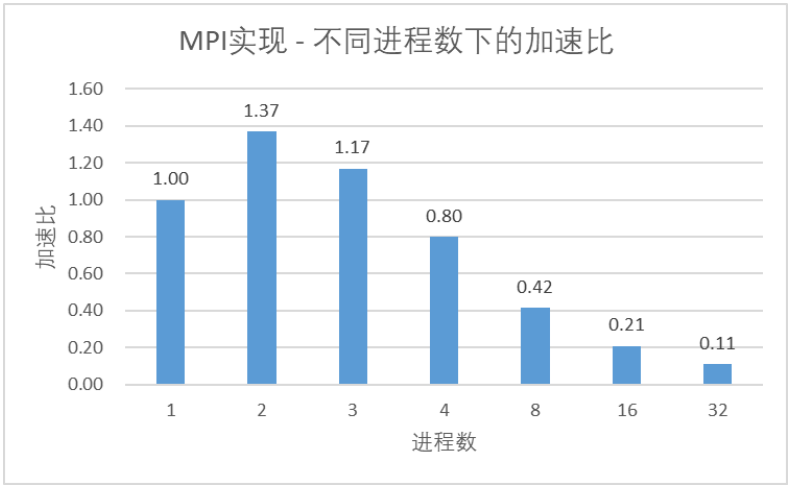
使用如下命令编译源代码：
mpicc -o main main.c MyUtils.c
再按如下格式运行可执行文件：
mpirun -np 2 ./main ./test/ dict.txt result.txt
说明：①"2"表示进程个数，最小为 2；②"./test/"代表文档所在目录，末尾需要有'/'）；③" dict.txt "代表字典文件路径；④"result.txt"代表结果文件路径。

程序会在终端打印输出运行时间，且生成存放文档向量信息结果文件。

为减少误差，这里使用了 shell 脚本进行 10 轮实验（程序同目录下，终端输入"./run.sh"即可执行脚本）。由于这里的 MPI 程序采用了管理者/工人模式，实际“干活”的是工人进程，所以这里以工人进程的个数来进行实验，计算加速比。每一轮中，（工人）进程数分别取 1、2、3、4、8、16 和 32。最终，可得到数据如下表：

进程数（工人进程）		1	2	3	4	8	16	32
执行时间（单位：s）	Round 1	0.165	0.111	0.144	0.168	0.341	0.665	1.697
	Round 2	0.171	0.106	0.111	0.149	0.400	0.764	1.434
	Round 3	0.141	0.101	0.145	0.144	0.388	0.761	1.321
	Round 4	0.156	0.128	0.116	0.136	0.364	0.787	1.098
	Round 5	0.138	0.113	0.115	0.256	0.343	0.674	1.440
	Round 6	0.130	0.105	0.138	0.225	0.335	0.694	1.191
	Round 7	0.136	0.117	0.105	0.209	0.299	0.643	1.402
	Round 8	0.156	0.106	0.141	0.234	0.428	0.917	1.411
	Round 9	0.164	0.102	0.133	0.127	0.335	0.629	1.166
	Round 10	0.135	0.102	0.128	0.223	0.360	0.605	1.262
平均时间		0.149	0.109	0.128	0.187	0.359	0.714	1.342
加速比		**1.00**	**1.37**	**1.17**	**0.80**	**0.42**	**0.21**	**0.11**

将加速比绘制成柱状图如下：



3. OpenMP 实现运行结果

使用如下命令编译源代码：

gcc -fopenmp -o main main.c MyUtils.c

再按如下格式运行可执行文件：

./main 4 ./test/ dict.txt result.txt

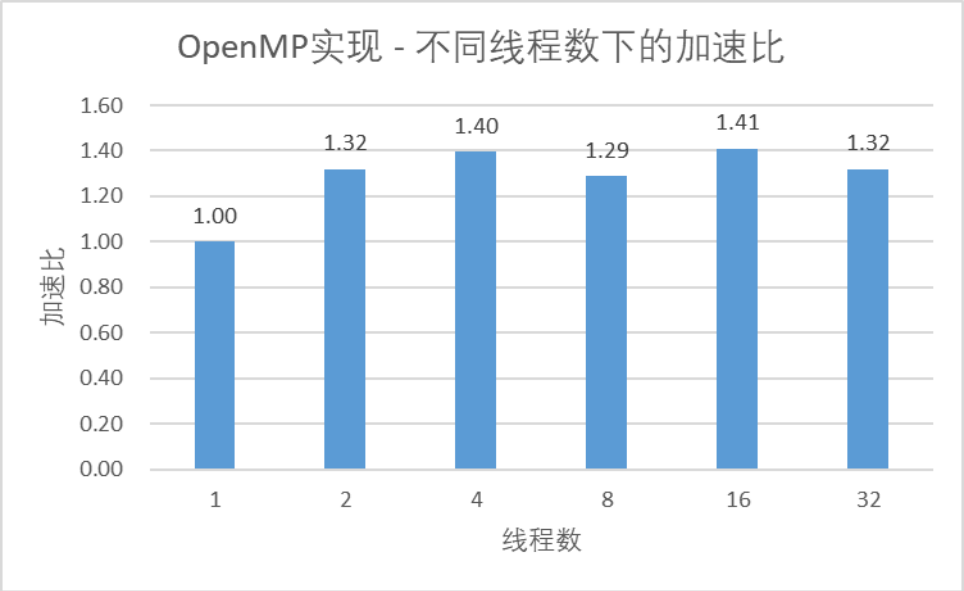
说明：①"4"表示进程个数；②"./test/"代表文档所在目录，末尾需要有'/')；③" dict.txt "代表字典文件路径；④"result.txt"代表结果文件路径。

程序会在终端打印输出运行时间，且生成存放文档向量信息结果文件。

为减少误差，这里使用了 shell 脚本进行 10 轮实验（程序同目录下，终端输入"./run.sh"即可执行脚本）。每一轮中，线程数分别取 1、2、4、8、16 和 32。最终，可得到数据如下表：

线程数		1	2	4	8	16	32
执行时间（单位：s）	Round 1	0.104	0.082	0.077	0.088	0.073	0.073
	Round 2	0.102	0.078	0.069	0.082	0.082	0.111
	Round 3	0.122	0.082	0.077	0.085	0.076	0.079
	Round 4	0.108	0.078	0.077	0.090	0.075	0.076
	Round 5	0.111	0.085	0.077	0.091	0.072	0.083
	Round 6	0.102	0.081	0.096	0.079	0.095	0.085
	Round 7	0.104	0.079	0.088	0.077	0.075	0.073
	Round 8	0.103	0.084	0.066	0.082	0.068	0.075
	Round 9	0.119	0.079	0.069	0.078	0.067	0.078
	Round 10	0.091	0.079	0.067	0.074	0.072	0.075
平均时间		0.107	0.081	0.076	0.083	0.076	0.081
加速比		**1.00**	**1.32**	**1.40**	**1.29**	**1.41**	**1.32**

将加速比绘制成柱状图如下：



4. 结果分析

本次实验中，由于采用的 Hash 方法较为高效，所以总的计算时间不是很大，总体来看加速比没有非常接近 2，并行额外开销相比之下比较大。这一点在 MPI 实现中有很大的体现。

MPI 实现的实验结果符合预期，即：随着进程数增加，加速比逐渐增大到最大值，再随着进程数的增大，加速比下降。当进程数增加，越来越充分利用 2 个物理 CPU 核的并行资源，加速比也就接近于最高。而进程数增加到一定程度，物理 CPU 核有限的，再增加线程数也无法从物理上增加程序运行的并行程度，反而因为进程数的增加，带来进程通信和管理等的额外开销，所以使得运行时间增加，加速比下降。

同时，后期下降的幅度很大，显示出增加进程带来的额外开销增长较为明显。在 MPI 程序中，进程通信的开销主要体现在工人进程之间广播字典文件内容，字典文件大小为 38.9k，广播这一内容无疑是需要很大的开销。而 Bcast 的开销与进程个数是线性关系，当进程大到一定程度后，这一开销成为主要耗时。进程数从 4、8、16 到 32 时，加速比每次刚好差不多是前一次的一半，正好可以验证这一想法。

而 OpenMP 是利用多线程来加速程序执行，在共享内存系统上运行。而线程与进程对比下，线程是更轻量级的，可以提供更细粒度的并行，且通信等的额外开销更小（线程不用完全复制出一整个独立的内存空间，多个线程可以共用一个；进程则是每个进程有自己独立的内存空间）。而本次实验中 OpenMP 程序没有涉及线程间数据通信，所以自然结果中 OpenMP 显示出的并行额外开销小，即随着线程增大，加速比没有明显下降。

三、 实验中遇到的问题及解决办法

1. MPI 进程个数随机

每次生成的进程个数随机，比如指定 5 个进程时，虽然 MPI_Comm_size()返回的是 5，但是如果每个进程都 printf 输出，每次是 1~5 随机个数的进程会执行输出。

解决：

检查和比对 MPI 程序代码结构是否完整，发现结尾处缺少"MPI_Finalize();"，补上之后即程序正常。

2. 处理字典内容时出现问题

在生成 Hash 表的函数 make_dict_Hash 中，使用了"strtok(dict_buffer, " \n\t");"这一函数对字典内容按照空白符进行分割，分割出一个个的单词。直接输出分割出的单词显示正常，但是计算文档向量时，没有按照预期计数。

解决：

使用 strcmp 将分割出的单词与指定的字符串比较（比如"a"），发现匹配失效，但是 printf 打印出来看起来是一样的。所以猜测可能是存在不可见字符，通过 strlen 查看长度发现果然长度不对。之后输出字符串中 ASCII 值，发现分割出的单词末尾都会多出一个 ASCII 值为 13 的字符，即'\r'。所以，这是由于不同平台文档换行回车格式不同导致的，所以 strtok 函数的第二个参数应该为" \r\n\t"。

3. **文件名顺序问题**

在 get_names 函数中，获取该目录下的所有文件的文件名之后，发现是乱序的。查阅相关博客后，得知是因为 readdir 的读取顺序，是按照在磁盘的索引顺序 d_off 来排序的。

解决：

使用库函数 qsort 来对存储文件名的数组排序。自定义排序规则：先按字符串长度排序，长度相等情况下，调用 strcmp 进行排序。这样得出的文件名顺序就是按照 doc0.txt、doc1.txt、doc2.txt、……、doc99.txt 的顺序。

四、 实验总结

- OpenMP 很多时候简单高效。特别适合在需要共享内存场景下，处理计算主体 for 循环部分的并行。使用 OpenMP 可以较为方便地从串行代码中修改得到并行代码，但是操作的灵活性不如 MPI，会有更多的局限性，适用面不如 MPI 广，比如无法在计算机集群上使用。

- MPI 程序中，在进程数增大到一定程度时，我们需要考虑到进程通信带来额外开销的影响。比如本次实验中，计算处理时间不长（因为使用的哈希效率较高），但是相比之下并行额外开销较大。

- 在本次实验中，虽然 OpenMP 执行效果和加速效果都好于 MPI，但这并不说明本次实验中的 OpenMP 完胜 MPI。它们各自适用的地方有不同。比如，这里的 OpenMP 实现在负载均衡方面做得就不够好，没有采用灵活的任务划分方式，而是一开始就指定文档给对应线程。如果文档之间大小差异较大，就可能出现有些线程早早就完成任务，从而空闲，这种情况下就会需要特别考虑负载均衡。所以 MPI 实现中的管理者/工人模式则变得非常有用。如果处理的总数据量很大，那么 MPI 中的并行额外相对就较小，而负载均衡的重要性就显现出来了，所以那种场景下 MPI 程序的表现可能会更加优秀。

五、 参考文献及资料

[1] Quinn M，奎因，陈文光, et al. MPI 与 OpenMP 并行程序设计:C 语言版[M]. 清华大学出版社, 2004.

[2] C 语言中的 dirent.h 说明_hello188988 的专栏-CSDN 博客. https://blog.csdn.net/hello188988/article/details/47711217

[3] Linux 上 readdir 遍历文件夹按文件名排序_krens 的专栏-CSDN 博客_readdir 函数按时间排序. https://blog.csdn.net/krens/article/details/78365382?utm_source=blogxgwz4