LZ77 压缩算法并行

李瑞鹏，师浩然

June 28, 2017

Abstract

我们先选取了串行的 LZ77 算法，并介绍了其背景。接着分析了并行算法的实现与并行性。最后我们论述了实验的并行效果。

1 背景介绍

1.1 Lempel-Ziv 编码

在大数据时代，数据压缩技术有很广泛的应用。对于我们传输的信息，信息熵通常较低，为了使数据能进一步压缩，我们需要增大信息熵。而我们对于信源编码有一类非常流行的技术，而且是通用最优的（即对于任意平稳遍历信源，渐近压缩率接近信源的熵率），而且容易实现。这类算法称为 Lempel-Ziv 算法，以两篇开创性的论文作者命名。这些算法也称为自适应字典式压缩算法。通常历史上，我们有 LZ77（滑动窗 Lempel-Ziv 算法）与 LZ78（树结构 Lempel-Ziv 算法），有时他们各自分别简称为 LZ1 与 LZ2. 在这里我们针对 LZ77 算法进行并行设计。

然而随着处理器向多核和并行发展的趋势，我们需要对原来的串行算法做一些改进，以提高算法的执行效率。目前主流压缩软件也多用基于 LZ77 的压缩算法，基本都基于 CPU 多线程并行压缩，但还没有更多的采用 OpenCL 等 GPU 多核异构结构进行并行加速。我们在这里也进行了一些探索。

在本项目中，我们尝试将经典的 LZ77 压缩算法进行并行尝试，通过 OpenMP 的技术在多核 CPU 上进行实现。在压缩算法的实现中，我们需要达到执行时间和压缩率达到一个 trade-off，并达到一个尽量高的加速比。接下来我们将介绍下 LZ77 的算法流程，再介绍并行处理，最后讨论一下并行的效果。

1.2 LZ77 串行算法

这个算法是在 1977 年的文章提出的算法，其主要思想是在一个过去字符窗口的任何地方通过查找最长匹配进行字符串编码，同时利用指向窗中匹配位置和匹配长度的指针表示字符串。这个基本算法有很多变式，我们采用的是 Storer 和 Szymanski 给出的描述。

假定有限字母表的字符串 $x_1, x_2, \cdots$ 需要被压缩。字符串 $x_1, x_2, \cdots, x_n$ 的解析 S 是将该字符串划分为若干词组，用逗号隔开。设 W 为窗口的长度。此时算法描述如下：假定已经将字符串压缩到时刻 $i-1$, 然后，为了找到下一个词组，先计算最大的 k，使得对某个 j，$i-1-W \leq j \leq i-1$, 长度为 k 并起始于 x_j 的字符串等于起始于 x_j 的字符串（长度为 k）（即对任意的 $0 \leq l < k$, 有 $x_{j+l} = x_{i+l}$). 于是，下一个词组的长度为 k（即 $x_i \cdots x_{i+k-1}$ ）, 且表示为二元对（P,L），其中 P 为匹配的起始位置，L 为匹配的长度。如果窗口中没有找到匹配，则下一个字符将无压缩地被发送。为区分这两种情形，需要一个标识位。因此词组有两种类型:（F,P,L）或（F,C），其中 C 表示未压缩的字符。

注意,（指针，长度）对的目标表示可能延伸超出窗口，从而导致与新的词组重叠。在理论上，这样的匹配可以任意长。而在实际中，最大词组长度限制为不能超过某个参数。

这个算法好比使用了一个字典，它由窗中字符串的所有子串与所有单字符构成。算法是要找到字典内的最长匹配，并且分配一个指针给这个匹配。而且这个 LZ77 算法被证明为了渐进最优的，gzip 和 pkzip 都是 LZ77 这个版本。

Algorithm 1 LZ77 算法

```
1:  begin
2:     fill view from input
3:     while (view is not empty) do
4:     begin
5:        find longest prefix p of view starting in coded part
6:        i := position of p in window
7:        j := length of p
8:        X := first char after p in view
9:        output(i,j,X)
10:       add j+1 chars
11:    end
12: end
```

2 并行算法

2.1 并行思路

本实验中使用 LZ77 算法对文件进行压缩。我们并行的思路也是相当简单，就是通过牺牲部分压缩率，来提升执行时间，在其中找到一个 trade-off. 首先，我们对需要压缩的文件进行一个拆分，然后利用并行对子文件进行压缩，再将文件进行合并，得到最后的压缩文件。而解压则是一个更加顺利的过程，将压缩后的文件分成多个子文件，再分别压缩，最后合成原来的文件，从而得到正确的原文件。

2.2 并行算法实现细节

1. Naive LZ77: 通过 Hash 表表示字典 + 朴素字符串匹配，实现 LZ77 算法

2. OpenMP: 在 LZ77 Hash 的基础上进行 OpenMP 进行并行处理, 将文件分拆

数据源：通过 generator 生成若干连续的数字，来进行压缩、解压

2.3 并行算法分析

LZ77 压缩算法主要核心部分是字符串比较，用 OpenMP 是比较合适的，是一个 SPMD 的模型。因此在这里分割文件，并匹配前 n 位字符，就可以达到一个较为不错的并行速度，通过选取合适的线程数，可以很好的实现执行时间和压缩率的一个权衡。

同时，据文献查询，在 GPU 上实现 LZ77 数据压缩并不是一个好的选择，因为 LZ77 算法含大量的内存比较和 I/O 操作，不符合 GPU 的架构特性，所以在 GPU 上执行数据压缩的速度不如 CPU 上，而增加 CPU 和 GPU 间的内存带宽是更好发挥 GPU 特性的一种方法。

3 实验结果

在测试环节，我们采用文本文件进行测试。测试平台为 Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz,cache size : 25600 KB, 测试结果如下：

数据 \ 项目 / 线程	时间	加速比	效率
1	122.95	1	1
2	52.94	2.32	1.16
4	27.07	4.54	1.14
8	14.92	8.24	1.03

我们可以看到采用 OpenMP 加速有明显的效果，同时达到了很高的效率。我们可以看到，吞吐量相比串行程序是有大幅提升的，我们认为基于 OpenMP 的实现还是能很高效地实现并行的, 这是基于 CPU 多线程的高效 IO 处理能力, 使得吞吐量得到大幅度提升。

但压缩率来说，并没有明显的大幅提升，基本上对于重复率较高的文本文件，几乎能达到 50% 的压缩率，对比起商用软件可以达到的 40% 压缩率还是有待提升的。但如果对稠密的二进制文件基本压缩率都是相当低的。同时由于引入一些附加信息，压缩文件会比原来更大一些。这还是有相当多的其他路径可以进行处理，比如提升算法的精细程度，或者引入 Huffman 编码。

References

[1] Gordon Cockburn and Adrian John Hawes. Method and arrangement for data compression according to the lz77 algorithm, 2007.

[2] Thomas M Cover and Joy A Thomas. *Elements of information theory /*. John Wiley, 2003.