



# 并行程序设计-字符串匹配

---



# 目 录

- 1 功 能 描 述
- 2 单 线 程 算 法
- 3 多 线 程 算 法
- 4 实 验 设 计
- 5 实 验 结 果

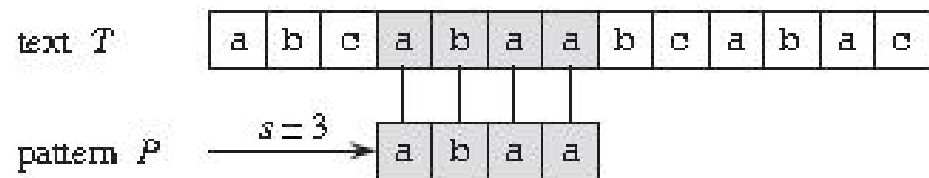


1

# 功能描述

---

# 字符串匹配问题



**文本** (Text) 是一个长度为  $n$  的数组  $T[1..n]$ ;

**模式** (Pattern) 是一个长度为  $m$  且  $m \leq n$  的数组  $P[1..m]$ ;

$T$  和  $P$  中的元素都属于有限的字母表  $\Sigma$  表;

如果  $0 \leq s \leq n-m$ , 并且  $T[s+1..s+m] = P[1..m]$ , 即对  $1 \leq j \leq m$ , 有  $T[s+j] = P[j]$ , 则说模式  $P$  在文本  $T$  中出现且位移为  $s$ , 且称  $s$  是一个有效位移 (Valid Shift) 。



# 字符串匹配问题

具体来说，我们需要一个函数：

```
auto search(const char* text, const size_t text_len, const char* pattern, const size_t pattern_len)  
    -> std::vector<size_t>;
```

函数通过给定的文本 `text` 和模式串 `pattern`，返回 `text` 中出现的模式串的位置列表。



# 2

## 单 线 程 算 法

---



# 常用的字符串匹配算法

## 朴素算法 (Naive Algorithm)

该算法的基本思想是从主串  $T$  的第一个字符开始和模式串  $P$  的第一个字符进行比较，若相等，则继续比较二者的后续字符；否则，模式串  $P$  回退到第一个字符，重新和主串  $T$  的第二个字符进行比较。如此往复，直到  $T$  或  $P$  中所有字符比较完毕。

## Knuth-Morris-Pratt 算法 (KMP Algorithm)

KMP算法的思想是，当子串与模式串  $P$  不匹配时，其实已经知道了前面已经匹配成功那一部分的字符（包括子串与模式串），因此可以确定下一步从哪里开始，从而避免重新检查先前匹配的字符。

算法分为两个部分，第一部分是在预处理阶段处理模式串  $P$ ，构建 `next` 数组，复杂度为  $O(n)$ ，其中  $n$  是模式串的长度；第二部分就是匹配过程了，复杂度是  $O(m)$ ，其中  $m$  是文本串的长度。

# KMP (Knuth-Morris-Pratt)

```
1 static void build_prefix_suffix_array(const char* pattern, size_t  
pattern_len, int* pps) {  
2     int length = 0;  
3     pps[0] = 0;  
4     size_t i = 1;  
5     while (i < pattern_len) {  
6         if (pattern[i] == pattern[length]) {  
7             length++;  
8             pps[i] = length;  
9             i++;  
10        }  
11        else {  
12            if (length != 0)  
13                length = pps[length - 1];  
14            else {  
15                pps[i] = 0;  
16                i++;  
17            }  
18        }  
19    }  
20 }
```

```
1 auto kmp_search(const char* text, const size_t text_len, const char* pattern,  
const size_t pattern_len) -> std::vector<size_t> {  
2     // 部分匹配表 (Partial Match Table), 也称为前缀后缀表 (Prefix-Suffix  
Table).  
3     int pps[pattern_len];  
4     build_prefix_suffix_array(pattern, pattern_len, pps);  
5  
6     std::vector<size_t> result;  
7     // i用于遍历文本, j用于遍历模式串。  
8     int i = 0;  
9     int j = 0;  
10    while (i < text_len) {  
11        // 如果当前字符匹配成功, 则模式串和文本都向后移动一个字符。  
12        if (pattern[j] == text[i]) {  
13            j++;  
14            i++;  
15        }  
16        // 完整匹配, 将当前匹配的起始索引加入结果。  
17        if (j == pattern_len) {  
18            result.push_back(i - j);  
19            // 根据部分匹配表调整模式串指针j。  
20            j = pps[j - 1];  
21        }  
22        // 如果字符不匹配, 并且i没有到达文本尾部。  
23        else if (i < text_len && pattern[j] != text[i]) {  
24            // j不为0时根据部分匹配表回溯。  
25            // 不是从模式串的开始位置重新匹配, j回到有最大前缀后缀匹配长度的位置。  
26            if (j != 0)  
27                j = pps[j - 1];  
28            // j为0时, 则移动文本指针i。  
29            else  
30                i = i + 1;  
31        }  
32    }  
33    return result;  
34 }  
35 }
```



## SIMD-based 搜索

假设我们有一些 8-byte 的寄存器，我们要在字符串 “a\_cat\_tries” 中搜索子串 “cat”。首先我们将 “cat” 的第一个字节和最后一个字节填充到两个寄存器中，并尽可能地重复知道寄存器被填满。

$$F = [ c \mid c \mid c \mid c \mid c \mid c \mid c \mid c ]$$
$$L = [ t \mid t \mid t \mid t \mid t \mid t \mid t \mid t ]$$

然后将字符串 “a\_cat\_tries” 加载到另外两个寄存器中，其中一个寄存器从第二个字符开始加载。

$$A = [ a \mid \_ \mid c \mid a \mid t \mid \_ \mid t \mid r ]$$
$$B = [ c \mid a \mid t \mid \_ \mid t \mid r \mid i \mid e ]$$

然后比较两组寄存器的内容，对应位置的内容相同为 1，反之则为 0。

$$AF = (A == F) = [ 0 \mid 0 \mid 1 \mid 0 \mid 0 \mid 0 \mid 0 \mid 0 ]$$
$$BL = (B == L) = [ 0 \mid 0 \mid 1 \mid 0 \mid 1 \mid 0 \mid 0 \mid 0 ]$$

最后我们将两个寄存器的内容合并，即 “位与” 运算。

$$\text{mask} = [ 0 \mid 0 \mid 1 \mid 0 \mid 0 \mid 0 \mid 0 \mid 0 ]$$

此时我们发现只有第 2 位（从零开始）是 1，这说明只有从此处开始搜索才有可能搜索到子串。这就减少了我们的搜索次数，

# SIMD-based



```
1 auto simd_search(const char* text, const size_t text_len, const char* pattern, const size_t pattern_len) ->
  std::vector<size_t> {
2     std::vector<size_t> result;
3
4     // 向寄存器中填充 needle 的第一个字节
5     const __m256i first = _mm256_set1_epi8(pattern[0]);
6     // 向寄存器中填充 needle 的最后一个字节
7     const __m256i last = _mm256_set1_epi8(pattern[pattern_len - 1]);
8
9     for (size_t i = 0; i < text_len; i += 32) {
10
11         // 向寄存器中填充 s 的部分内容
12         const __m256i block_first = _mm256_loadu_si256(reinterpret_cast<const __m256i*>(text + i));
13
14         // 向寄存器中填充 s 的部分内容, 相对于上一行, 本次填充的内容有所偏移
15         const __m256i block_last = _mm256_loadu_si256(reinterpret_cast<const __m256i*>(text + i + pattern_len - 1));
16
17         // 比较两组寄存器
18         const __m256i eq_first = _mm256_cmpeq_epi8(first, block_first);
19         const __m256i eq_last = _mm256_cmpeq_epi8(last, block_last);
20
21         // 合并两个寄存器的比较结果
22         uint32_t mask = _mm256_movemask_epi8(_mm256_and_si256(eq_first, eq_last));
23
24         while (mask != 0) {
25             // 找到第一个值为 1 的 bit 的下标
26             const auto bitpos = bits::get_first_bit_set(mask);
27
28             if (memcmp(text + i + bitpos + 1, pattern + 1, pattern_len - 2) == 0) {
29                 result.push_back(i + bitpos);
30             }
31
32             mask = bits::clear_leftmost_set(mask);
33         }
34     }
35
36     return result;
37 }
```



# 多线程算法

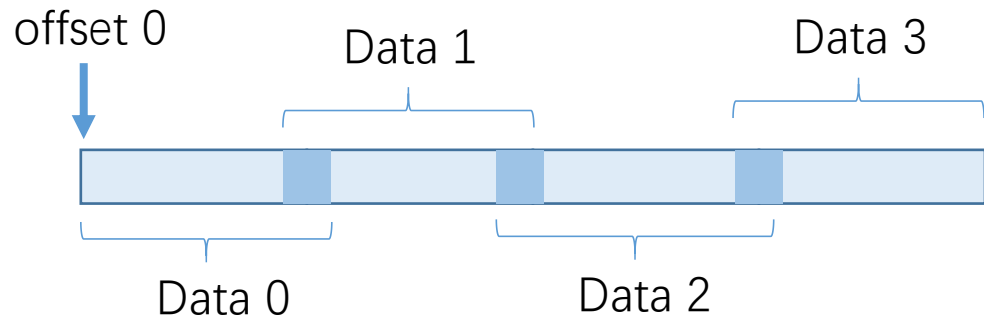
---

# 数据分配

```
1 struct Task {  
2     size_t offset;  
3     size_t size;  
4  
5     Task() = default;  
6     Task(size_t offset, size_t size): offset(offset), size(size) {}  
7 };
```

对于并行的字符串匹配，我们在启动线程前将空间分解为多个块（Data），并计算其起始偏移量和数据长度。

对于非第一个空间数据分配，它们的起始偏移量要向前一点点（ $\text{pattern\_len} - 1$ ），以保证搜索过程中不会有遗漏。





## 数据分配

```
// Generate tasks.  
// Assume that total size is 395, we split it into 4 tasks. And the length to the pattern is 5.  
// |__100__|__100__|__100__|__95__|  
// tasks are: [0, 100), [96, 200), [196, 300), [296, 395]  
auto base_addr = p;  
auto file_len = total_length;  
auto pattern_len = strlen(pattern);  
std::generate(tasks.begin(), tasks.end(), [&, i = 0]() mutable {  
    auto start = i == 0 ? 0: (i * task_size - (pattern_len - 1));  
    auto real_size = i == task_size - 1 ? std::min(task_size, file_len - i * task_size) : task_size;  
    real_size += pattern_len;  
    i++;  
    return Task(start, real_size);  
});
```

# KMP + OpenMP

```
// Assign tasks to threads.
#pragma omp parallel num_threads(threads)
{
    auto index = omp_get_thread_num();
    auto task = tasks[index];

    auto [_offset, _size] = task;
    auto result = kmp_search(reinterpret_cast<const char *>(base_addr + task.offset), _size, pattern, strlen(pattern));

    // kmp_search only returns the offset to the pattern from the start of the block, not the base address.
    for (auto &r: result) {
        r += _offset;
    }
    mid_result[index] = std::move(result);
}

auto end = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start).count();
std::cout << "task finished, costs " << std::dec << duration << " microseconds (" << display_time(duration) << ")" << std::endl;

auto result = std::vector<size_t>();
for (auto &r: mid_result) {
    result.insert(result.end(), r.begin(), r.end());
}

return {result, duration};
}
```

此处，kmp\_search 函数所返回的子串偏移量是相对于该任务的起始位置的，因此我们需要将结果换算成相对于整个查找区域的偏移量。



# 4

# 实 验 设 计

---



## 实验思路

1. 初始化：分配内存大小，并指定要搜索的字符串 PATTERN;
2. 在内存区域中随机放置模式字符串 PATTERN;
3. 使用单线程查找算法（KMP, SIMD-based search），并计时;
4. 使用基于 OpenMP 的改进算法，测试不同并行度下的用时;
5. 检查搜索结果，输出运行时间和加速比。





# 整体流程

```
int main() {
    const auto MIN_MEMORY_USE = 128 * 1024 * 1024L;
    const auto MAX_MEMORY_USE = 1 * 1024 * 1024 * 1024L;
    const auto PATTERN = "PATTERN";

    // 一次分配，多次使用，提高测试性能。
    auto p = new uint8_t[MAX_MEMORY_USE];
    memset(p, 0, MAX_MEMORY_USE); // 设置内存内容为 0

    // 内存大小
    for (auto size = MIN_MEMORY_USE; size <= MAX_MEMORY_USE; size *= 2) {

        generate_test_data(p, size, PATTERN, 5);
        auto durations = do_serial_test_in_memory(p, size, PATTERN);

        std::cout << "memory size: " << display_size(size) << ", serial & SIMD costs: ";
        for (auto duration: durations) {
            std::cout << display_time(duration) << ", ";
        }
        std::cout << ", parallel costs:";

        // 线程数
        for (auto cores = 2; cores <= 4; cores *= 2) {
            auto durations_with_threads = do_parallel_test_in_memory(p, size, PATTERN, cores);
            for (auto duration: durations) {
                std::cout << "c" << cores << ": " << display_time(duration) << ", ";
            }
            std::cout << std::endl;
        }
    }

    delete[] p;
    return 0;
}
```



# 测试数据的产生方法

我们希望尽量均匀地在存储空间中放置测试数据。不妨将可用内存分为若干块（block），在每个块中放置一个模式串。该方式可以加速测试数据的生成，并不影响其分布特征。

```
auto generate_test_data(uint8_t* base, size_t size, const char *pattern, size_t count) -> void {
    memclr(base, size);

    auto pattern_len = strlen(pattern);
    std::random_device rd; // 生成随机数
    std::mt19937 gen(rd()); // 伪随机数产生器
    // Designing a perfect placement algorithm can be complex. Here, we will simply evenly distribute the PATTERN
    // throughout the memory region.
    auto positions = std::vector<size_t>(count);
    auto block_size = size / count;
    std::uniform_int_distribution<int> dis(0, block_size - pattern_len); // 产生字符串, 类型为 int
    std::generate(positions.begin(), positions.end(), [&, i = 0, step = block_size]() mutable {
        return i++ * step + dis(gen);
    });

    auto i = 0;
    for (auto pos: positions) {
        auto _addr = base + pos;
        if (i++ < 4) {
            std::cout << "place pattern at 0x" << std::hex << reinterpret_cast<std::uintptr_t>(_addr) << std::endl;
        }
        memcpy(_addr, pattern, pattern_len);
    }

    auto flag = check_result_quickly(base, size, pattern, positions);
    if (!flag) {
        throw std::runtime_error("failed to place pattern in memory.");
    }
}
```



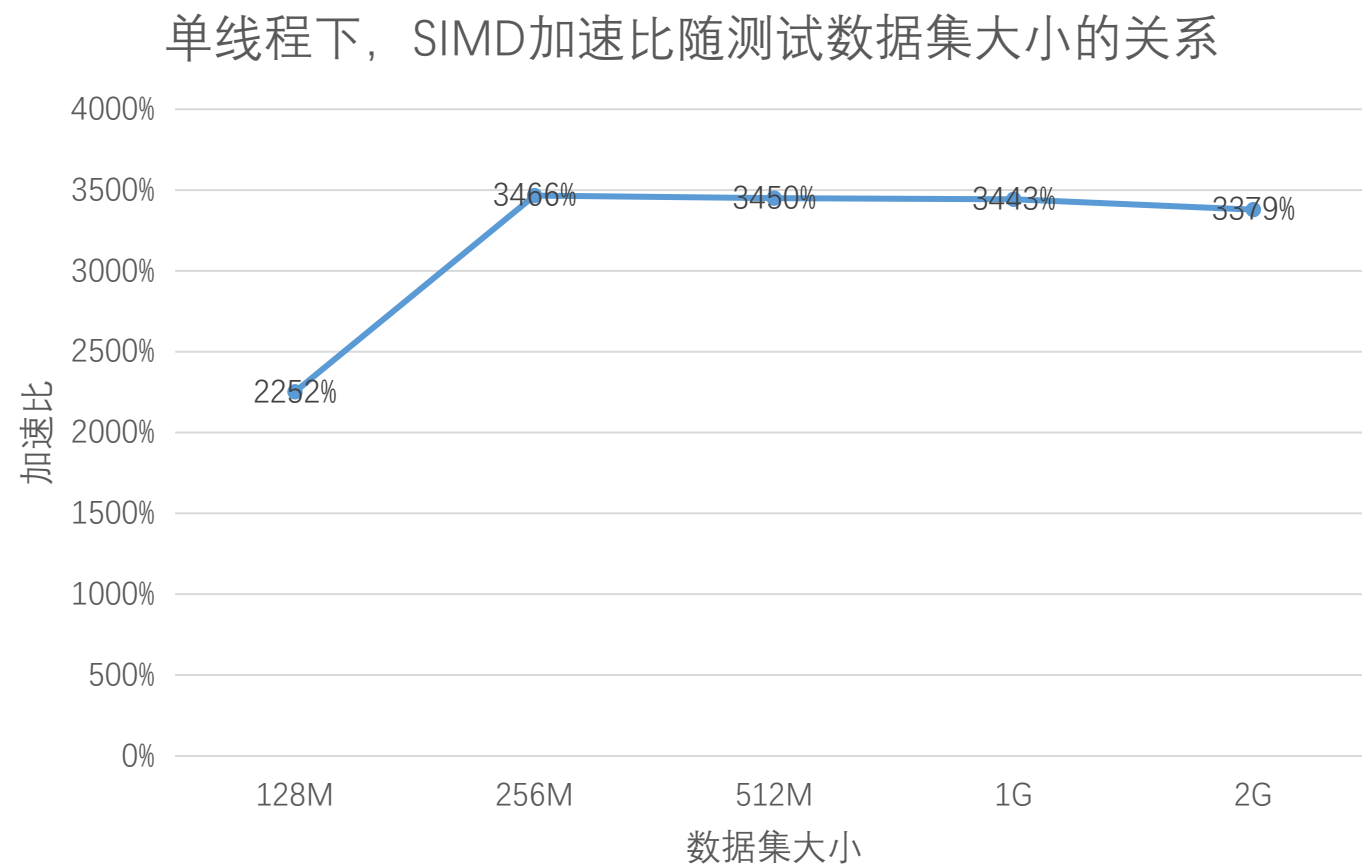
# 实 验 结 果

---



# 实验结果 单线程

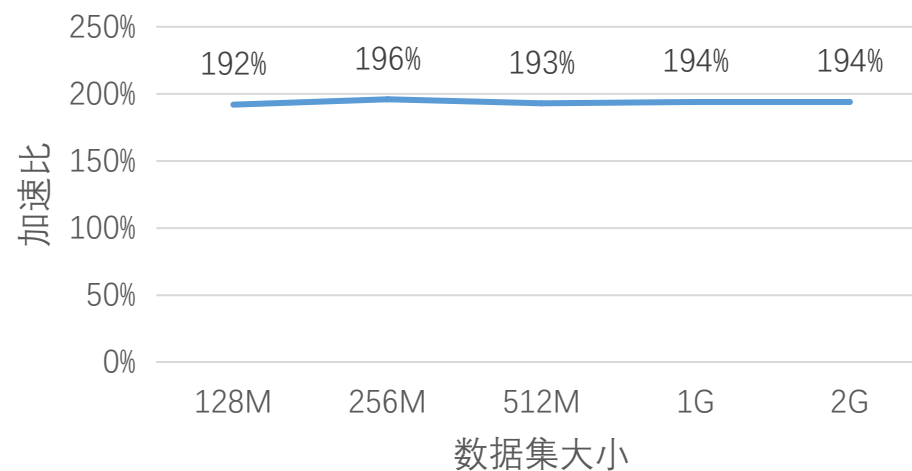
数据集大小	串行	SIMD	加速比
128M	187ms938us	8ms346us	2252%
256M	370ms955us	10ms702us	3466%
512M	736ms102us	21ms338us	3450%
1G	1s470ms319us	42ms710us	3443%
2G	2s940ms240us	87ms14us	3379%



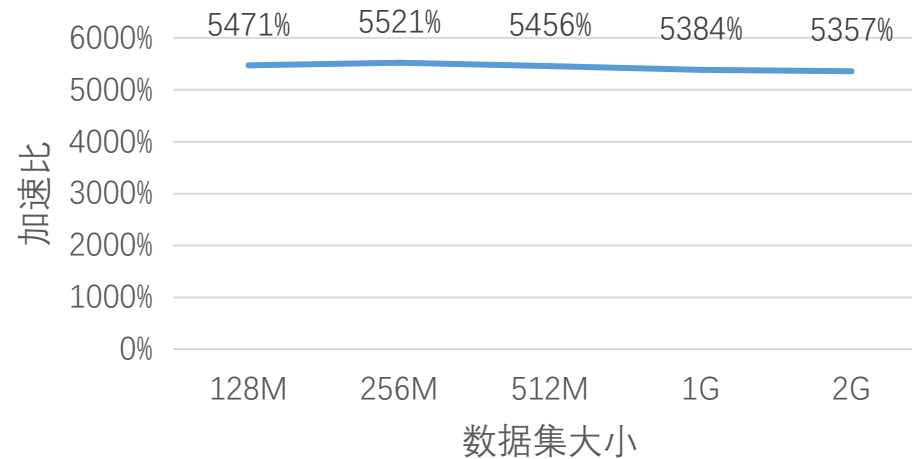
## 实验结果 多线程

数据集大小	OpenMP	OpenMP + SIMD	线程数
128M	97ms659us (192%)	3ms435us (5471%)	2
256M	189ms695us (196%)	6ms719us (5521%)	2
512M	382ms112us (193%)	13ms492us (5456%)	2
1G	756ms714us (194%)	27ms307us (5384%)	2
2G	1s511ms744us (194%)	54ms888us (5357%)	2

### OpenMP加速比 (2线程)



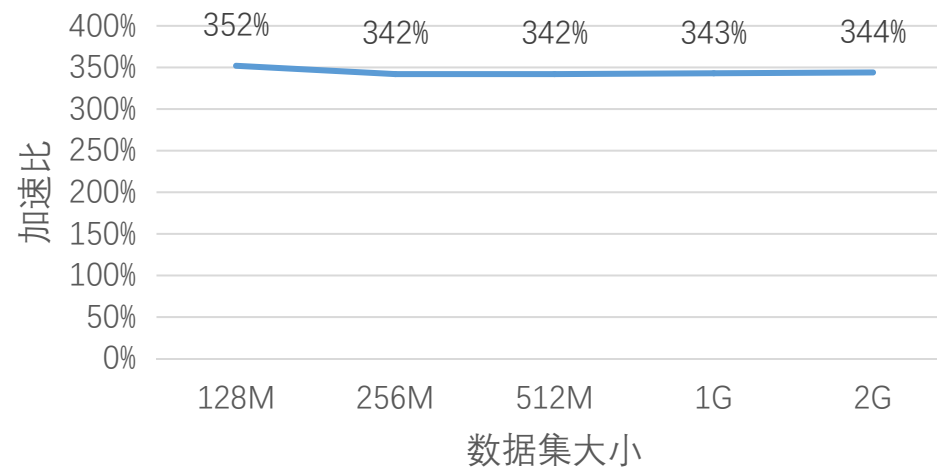
### OpenMP + SIMD加速比 (2线程)



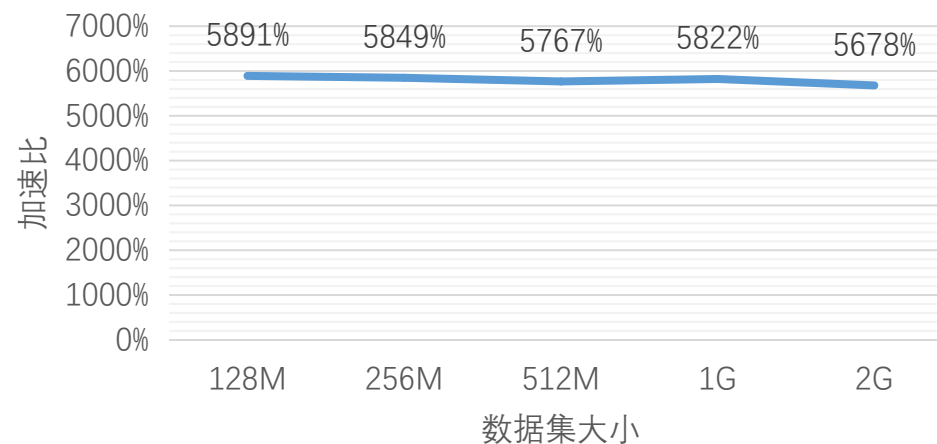
# 实验结果 多线程

数据集大小	OpenMP	OpenMP + SIMD	线程数
128M	53ms365us (352%)	3ms190us (5891%)	4
256M	108ms310us (342%)	6ms342us (5849%)	4
512M	215ms506us (342%)	12ms764us (5767%)	4
1G	428ms336us (343%)	25ms256us (5822%)	4
2G	853ms742us (344%)	51ms780us (5678%)	4

OpenMP的加速比（4线程）

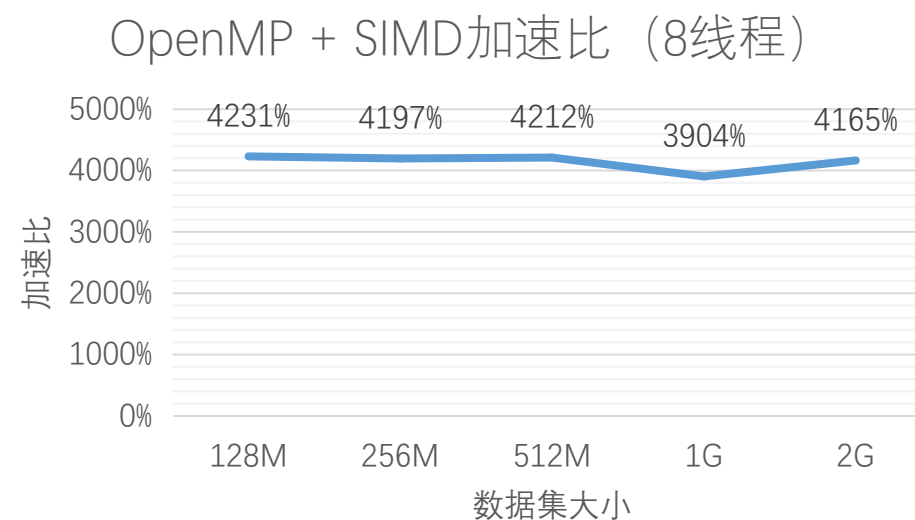
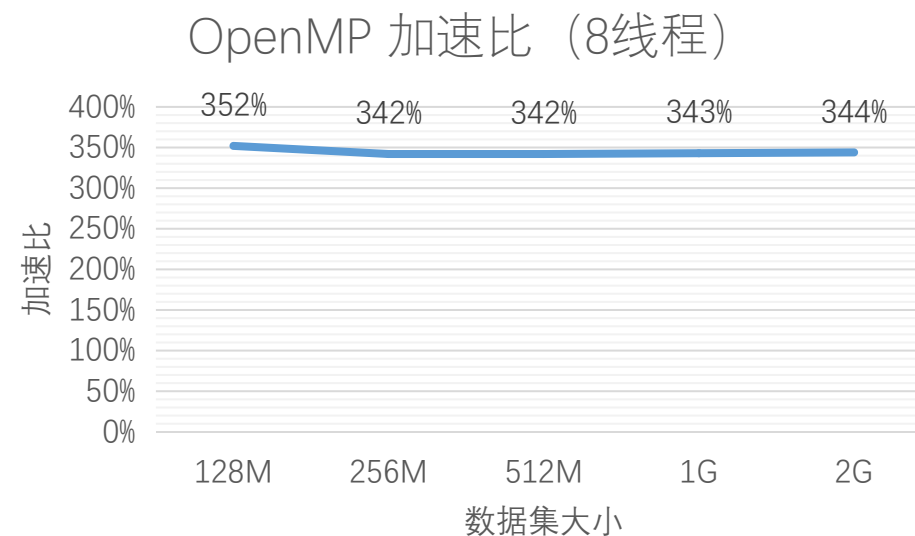


OpenMP + SIMD加速比（4线程）



# 实验结果 多线程

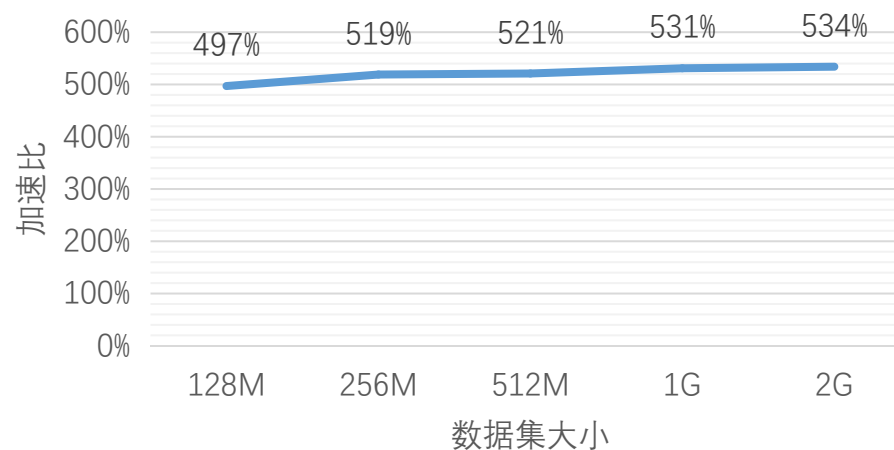
数据集大小	OpenMP	OpenMP + SIMD	线程数
128M	55ms653us (338%)	4ms442us (4231%)	8
256M	93ms683us (396%)	8ms838us (4197%)	8
512M	133ms54us (553%)	17ms476us (4212%)	8
1G	264ms390us (556%)	37ms664us (3904%)	8
2G	551ms360us (533%)	70ms592us (4165%)	8



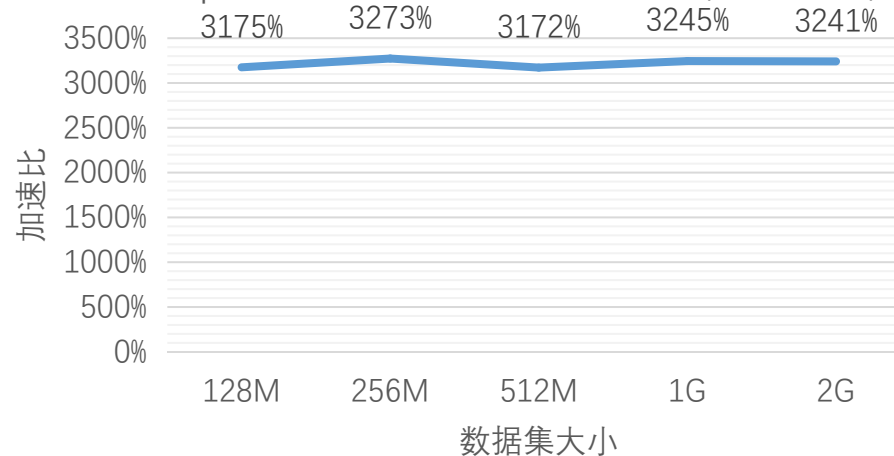
## 实验结果 多线程

数据集大小	OpenMP	OpenMP + SIMD	线程数
128M	37ms811us (497%)	5ms919us (3175%)	16
256M	71ms498us (519%)	11ms335us (3273%)	16
512M	141ms197us (521%)	23ms207us (3172%)	16
1G	277ms81us (531%)	45ms315us (3245%)	16
2G	550ms498us (534%)	90ms710us (3241%)	16

OpenMP 加速比 (16线程)



OpenMP + SIMD加速比 (16线程)

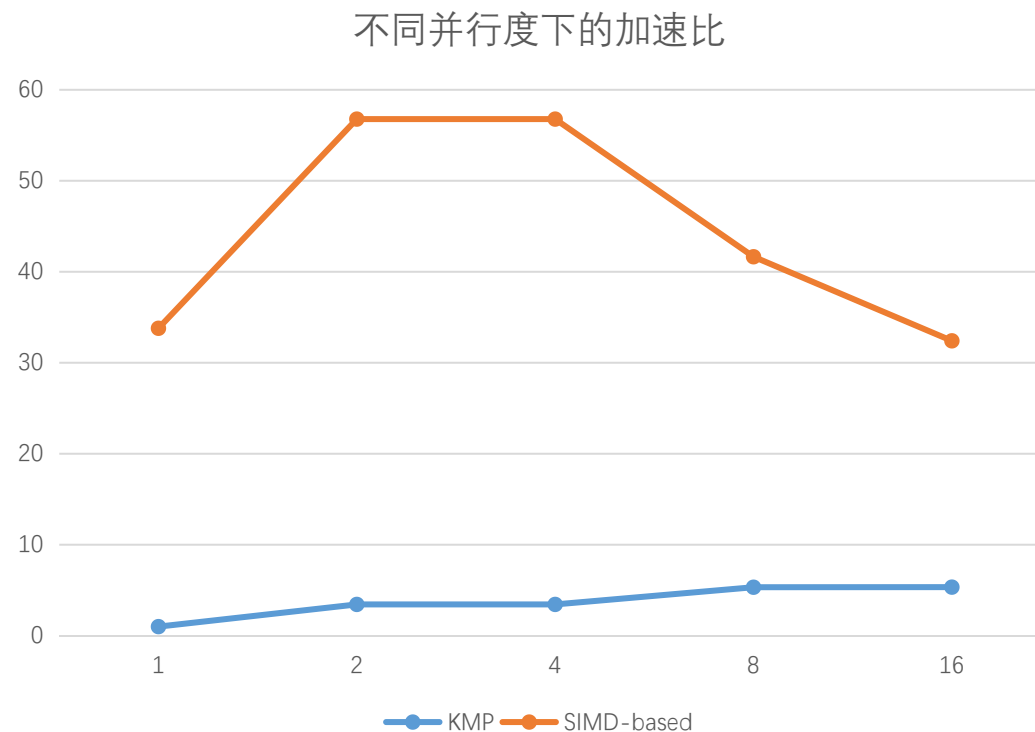




## 实验结果 不同并行度下的加速效果对比

数据集大小 2G

线程数	KMP	SIMD
1	2s940ms240us (100%)	87ms14us (3379%)
2	853ms742us (344%)	51ms780us (5678%)
4	853ms742us (344%)	51ms780us (5678%)
8	551ms360us (533%)	70ms592us (4165%)
16	550ms498us (534%)	90ms710us (3241%)





请 老 师 批 评 指 正