《并行程序设计》
实验报告

学号：201936380086

姓名：陈涵

班级：2019 级软件工程 1 班

2022/4/3

华南理工大学本科实验报告

课程名称 并行程序设计 成绩评定

实验项目名称 OpenMP 环境配置及实现所给算法的 OpenMP 并行化

指导教师 汤德佑 实验地点 B7-333

学生姓名 陈涵 学号 201936380086

学院 软件学院 系 软件工程 专业

实验时间 年 月 日 午～ 月 日 午

1、实验目标

（1）在 Clang/G++环境下配置 OpenMP 环境；

（2）对以下算法利用 OpenMP 实现并行化。

i. 矩阵-向量乘法;

ii 字符串分组算法;

2、串行程序代码(串行程序已经给出，请把主要代码部分摘抄在下面)

i. 矩阵-向量乘法

```
/*--------------------------------------------------------------------
 * Function:    Mat_vect_mult
 * Purpose:     Multiply a matrix by a vector
 * In args:     A: the matrix
 *              x: the vector being multiplied by A
 *              m: the number of rows in A and components in y
 *              n: the number of columns in A components in x
 * Out args:    y: the product vector Ax
 */
void Mat_vect_mult(
                    double    A[]   /* in    */,
                    double    x[]   /* in    */,
                    double    y[]   /* out */,
                    int       m     /* in    */,
                    int       n     /* in    */) {
    int i, j;

    for (i = 0; i < m; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
```

```
            y[i] += A[i*n+j]*x[j];
    }
}   /* Mat_vect_mult */
```

ii 字符串分组算法（比较哈希方法和排序方法）

```
//group strings with hash in unordered_map STL
std::unordered_map<std::string, int> work_hash(std::vector<std::string> &origin)
{
    std::unordered_map<std::string, int> final_ans;//for final answer
    for(auto &s:origin)
    {
        //put string into hash table
        if(final_ans.find(s) == final_ans.end())
            final_ans[s] = 1;
        else
            final_ans[s]++;
    }
    return final_ans;
}
//group strings with sort
std::vector<std::pair<std::string, int> >
    work_sort_multithread(std::vector<std::string> &origin)
{
    std::vector<std::pair<std::string, int> > final_ans_sort;//for final answer
    std::sort(origin.begin(), origin.end());

    //write answer into vector
    std::string str="\0";
    int cnt=0;
    for(auto &s:origin)
    {
        if(str == s)
            cnt++;
        else
        {
            if(str != "\0")
                final_ans_sort.push_back(std::make_pair(str, cnt));
            str = s;
            cnt = 1;
        }
    }
    if(str != "\0")
```

```
            final_ans_sort.push_back(std::make_pair(str, cnt));
        return final_ans_sort;
}
```

3、并行程序关键代码

i. 矩阵-向量乘法

```
/*--------------------------------------------------------------------
 * Function:    Mat_vect_mult
 * Purpose:     Multiply a matrix by a vector
 * In args:     A: the matrix
 *              x: the vector being multiplied by A
 *              m: the number of rows in A and components in y
 *              n: the number of columns in A components in x
 * Out args:    y: the product vector Ax
 */
void Mat_vect_mult(
                    double    A[]   /* in    */,
                    double    x[]   /* in    */,
                    double    y[]   /* out */,
                    int       m     /* in    */,
                    int       n     /* in    */) {
    int i, j;
    #pragma omp parallel for num_threads(N)
    for (i = 0; i < m; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i*n+j]*x[j];
    }
}   /* Mat_vect_mult */
```

ii 字符串分组算法

代码段 1 (直接分段排序)

```
//group strings with sort multi-thread
std::vector<std::pair<std::string, int> >
        work_sort_multithread(std::vector<std::string> &origin)
{
        std::vector<std::pair<std::string, int> > final_ans_sort;//for final answer

        //split by N threads
        //each thread process in[split_edge[i], split_edge[i+1])
```

```
int one_part = origin.size()/N;
int split_edge[2*(N+1)];
split_edge[0]=0;
split_edge[N]=origin.size();
for(int i=N+1;i<2*(N+1);i++)
{
	split_edge[i]=split_edge[N];
}
for(int i=1;i<N;i++)
{
	split_edge[i] = split_edge[i-1] + one_part;
}

//parallel sort
#pragma omp parallel for num_threads(N)
for(int i=0;i<N;i++)
{
	sort(origin.begin()+split_edge[i], origin.begin()+split_edge[i+1]);
}

//merge
for(int merge_size=1;merge_size<N;merge_size*=2)
{
	#pragma omp parallel for num_threads(N)
	for(int i=0;i<N;i+=2*merge_size)
	{
		std::vector<std::string> temp1(origin.begin()+split_edge[i],
				origin.begin()+split_edge[i+merge_size]);
		std::vector<std::string>
				temp2(origin.begin()+split_edge[i+merge_size],
				origin.begin()+split_edge[i+2*merge_size]);
		merge(temp1.begin(), temp1.end(), temp2.begin(), temp2.end(),
				origin.begin()+split_edge[i]);
	}
}

//write answer into map
std::string str="\0";
int cnt=0;
for(auto &s:origin)
{
	if(str == s)
	{
		cnt++;
```

```
            }
            else
            {
                if(str != "\0")
                {
                    final_ans_sort.push_back(std::make_pair(str, cnt));
                }
                str = s;
                cnt = 1;
            }
        }
        if(str != "\0")
        {
            final_ans_sort.push_back(std::make_pair(str, cnt));
        }
        return final_ans_sort;
}
```

代码段 2 (按首字母分组排序)

```
//group strings with split by prefix then sort multi-thread
std::vector<std::pair<std::string, int> >
        work_prefix_sort_multithread(std::vector<std::string> &origin)
{
        //split by prefix-one char
        std::vector<std::string> origin_split[N];
        for(auto &s:origin)
        {
            origin_split[s[0]%N].push_back(s);
        }

        //parallel sort
        #pragma omp parallel for num_threads(N)
        for(int i=0;i<N;i++)
        {
            sort(origin_split[i].begin(), origin_split[i].end());
        }

        std::vector<std::pair<std::string, int> > final_ans_prefix_sort;//for final answer
        final_ans_prefix_sort.reserve(origin.size());
        //write answer into map
        std::string str="\0";
        int cnt=0;
        for(int i=0;i<N;i++)
        {
            if(origin_split[i].empty()) continue;
```

```
            for(auto &s:origin_split[i])
            {
                if(str == s)
                {
                    cnt++;
                }
                else
                {
                    if(str != "\0")
                    {
                        final_ans_prefix_sort.push_back(std::make_pair(str, cnt));
                    }
                    str = s;
                    cnt = 1;
                }
            }
        }
        if(str != "\0")
        {
            final_ans_prefix_sort.push_back(std::make_pair(str, cnt));
        }
        return final_ans_prefix_sort;
}
```

代码段 3 (哈希分组)

```
//group strings with hash multi-thread
std::unordered_map<std::string, int>
        work_hash_multithread(std::vector<std::string> &origin)
{
        std::unordered_map<std::string, int> hashTable_split[N];//for each thread
        std::unordered_map<std::string, int> final_ans_hash;//for final answer

        //split by N threads
        //each thread process in[split_edge[i], split_edge[i+1])
        int one_part = origin.size()/N;
        int split_edge[N+1];
        split_edge[0]=0;
        split_edge[N]=origin.size();
        for(int i=1;i<N;i++)
        {
            split_edge[i] = split_edge[i-1] + one_part;
        }


        //parallel group
```

```
    #pragma omp parallel for num_threads(N)
    for(int i=0;i<N;i++)
    {
        hashTable_split[i].reserve(split_edge[i+1]-split_edge[i]);
        for(int j=split_edge[i];j<split_edge[i+1];j++)
        {
            if(hashTable_split[i].find(origin[j]) == hashTable_split[i].end())
            {
                hashTable_split[i][origin[j]] = 1;
            }
            else
            {
                hashTable_split[i][origin[j]]++;
            }
        }
    }

    //merge
    //final_ans_hash.reserve(origin.size());
    for(int i=0;i<N;i++)
    {
        final_ans_hash.insert(hashTable_split[i].begin(), hashTable_split[i].end());
    }
    return final_ans_hash;
}
```

代码段 4 (首字母分片二维哈希分组)

```
std::vector<std::unordered_map<std::string, unsigned int> > hash_split[63];
    void work()
    {
        for(unsigned int i=0;i<63;i++)
        {
            hash_split[i].resize(thread_using);
        }

        //read file into buffer
        std::ifstream fin(input_filename, std::ios::binary);
        if(fin.is_open()==false)
        {
            std::cout<<"Fail to open the file!\n";
            return;
        }
        std::vector<char> buf(fin.seekg(0, std::ios::end).tellg());
        fin.seekg(0, std::ios::beg).read(&buf[0],
                static_cast<std::streamsize>(buf.size()));
```

```
if(buf[buf.size()-1]!='\n') buf.push_back('\n');
fin.close();

//split buffer into N parts
uint64_t split_part[thread_using+1];
split_part[0]=0;
split_part[thread_using]=buf.size();
uint64_t each_part=buf.size()/thread_using;
for(unsigned int i=1;i<thread_using;i++)
{
	split_part[i]=split_part[i-1]+each_part;
}

for(unsigned int i=1;i<thread_using;i++)
{
	while(buf[split_part[i]]!='\n')
	{
		split_part[i]++;
	}
}

//multicore-hashing
#pragma omp parallel for num_threads(thread_using)
for(unsigned int i=0;i<thread_using;i++)
{
	std::string tmp;
	tmp.clear();
	for(uint64_t idx=split_part[i];idx<split_part[i+1];idx++)
	{
		if(buf[idx]=='\n')
		{

					if(hash_split[table_ctoi[tmp[0]]][i].find(tmp)=
					=hash_split[table_ctoi[tmp[0]]][i].end())
			{
				hash_split[table_ctoi[tmp[0]]][i][tmp]=1;
			}
			else
			{
				hash_split[table_ctoi[tmp[0]]][i][tmp]++;
			}
			tmp.clear();
		}
		else
```

```
                {
                    tmp.push_back(buf[idx]);
                }
            }
        }
        buf.clear();
        buf.shrink_to_fit();

        //merge
        #pragma omp parallel for num_threads(thread_using)
        for(unsigned int i=0;i<63;i++)
        {
            for(unsigned int t=1;t<thread_using;t++)
            {
                for(auto &it:hash_split[i][t])
                {
                    if(hash_split[i][0].find(it.first)==hash_split[i][0].end())
                    {
                        hash_split[i][0][it.first]=it.second;
                    }
                    else
                    {
                        hash_split[i][0][it.first]+=it.second;
                    }
                }
            }
        }
    }
```

4、性能分析

测试环境 1	
CPU	Intel 9880H 8-core 2.3GHz
Memory	DDR4-2400MHz Dual-Channel 32GB
compiler	Apple Clang-1300.0.29.30
disk	APPLE SSD AP2048M-2TB@PCIe 3.0 x4

测试环境 2	
CPU	AMD Ryzen 7 5700G 8-core 4.6GHz
Memory	DDR4-4533MHz Dual-Channel 32GB
compiler	G++-4.8.1(tdm64-2)
disk	Sansumg PM9A1-1TB@PCIe 3.0 x4

低中高词频测试数据集（根据助教提供的代码和参考参数生成）		
20M_low.txt	20M_mid.txt	20M_high.txt
20000000 1 5	20000000 1 15	20000000 1 50

4.1 比较直接分段快速排序后归并和按首字符分组后排序

测试环境	使用代码	数据集	直接分段排序	分组排序
测试环境 1	代码段 1、2	20M_mid.txt	simple sort	prefix split sort

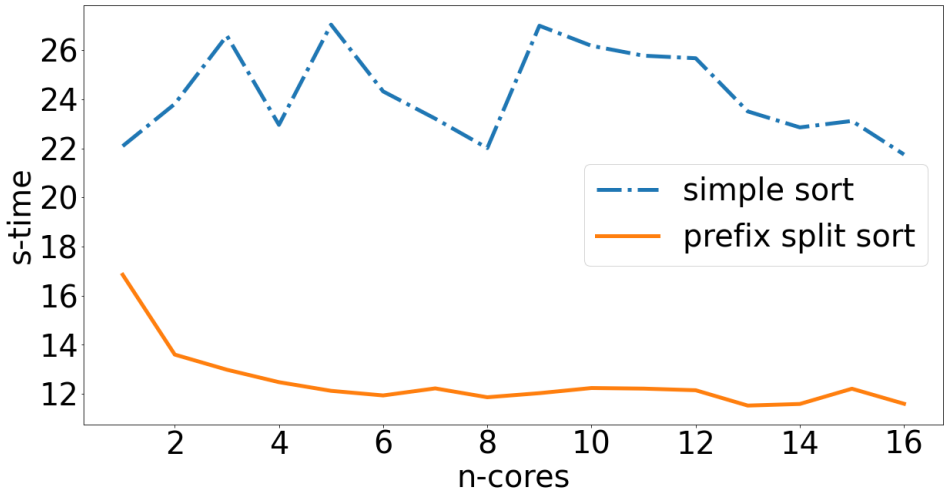


图 1 直接分段快速排序后归并和按首字符分组后排序用时对比

由于直接分段排序后，归并和去重部分只能串行处理，导致并行部分占比不高，用时远高于分组排序。同时直接分段排序在归并时采取两两归并的策略，在 2 的次幂线程数下用时低于其他线程数。

4.2 比较哈希分组和按首字母分组后排序

测试环境	使用代码	数据集	哈希方法	分组排序
测试环境 1	代码段 1、3	20M_low.txt	hash	split sort

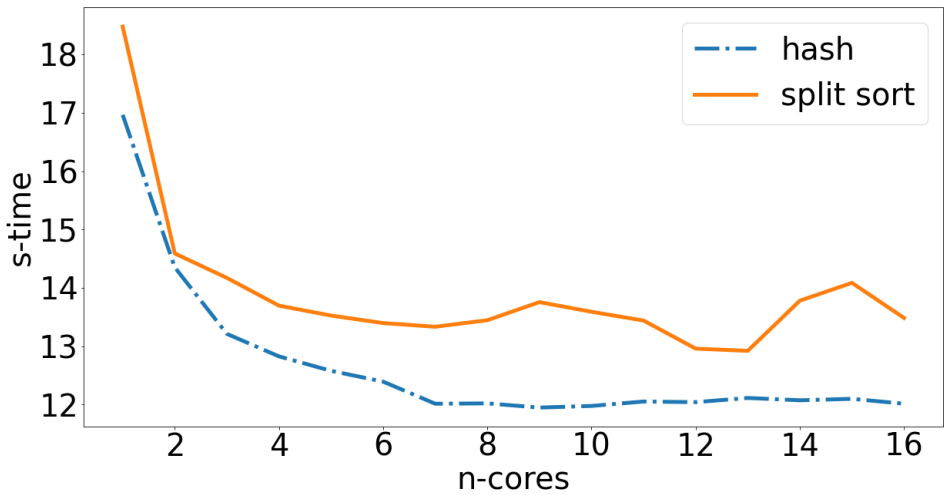


图 2 哈希分组和按首字母分组排序用时对比

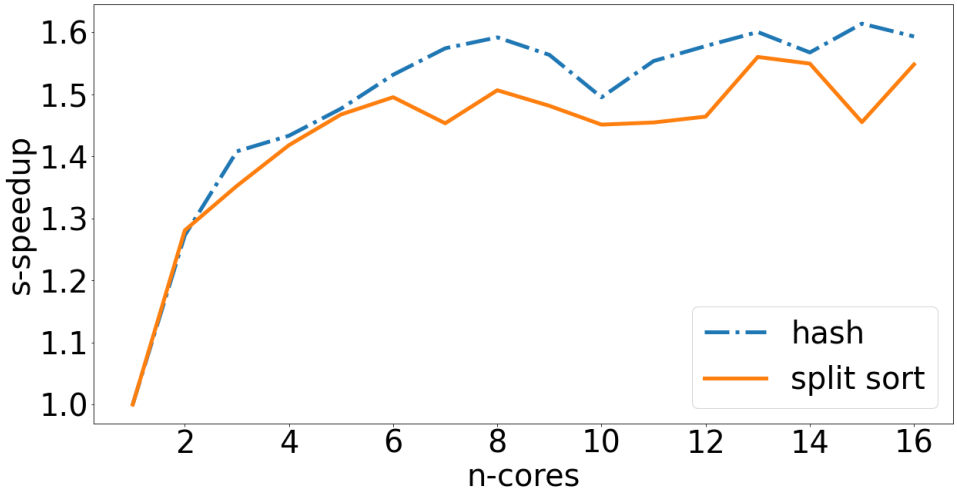


图 3 哈希与按首字母分组排序加速比对比

由于哈希不需要按首字母分组，减小了串行部分。同时，哈希方法不会重复储存相同的字符串，减小了内存开支，提高了缓存命中率和访存速度。可以猜测，哈希方法在高词频的数据集上会产生更好的效果。

由于测试环境 1 是笔记本，在高负载下会出现一定程度的降频，导致测试数据不稳定，接下来将在稳定环境下测试。

4.3 比较哈希分组在不同词频数据集下的效果

测试环境	使用代码	数据集	方法
测试环境 2	代码段 3	20M_low.txt, 20M_mid.txt, 20M_high.txt	hash

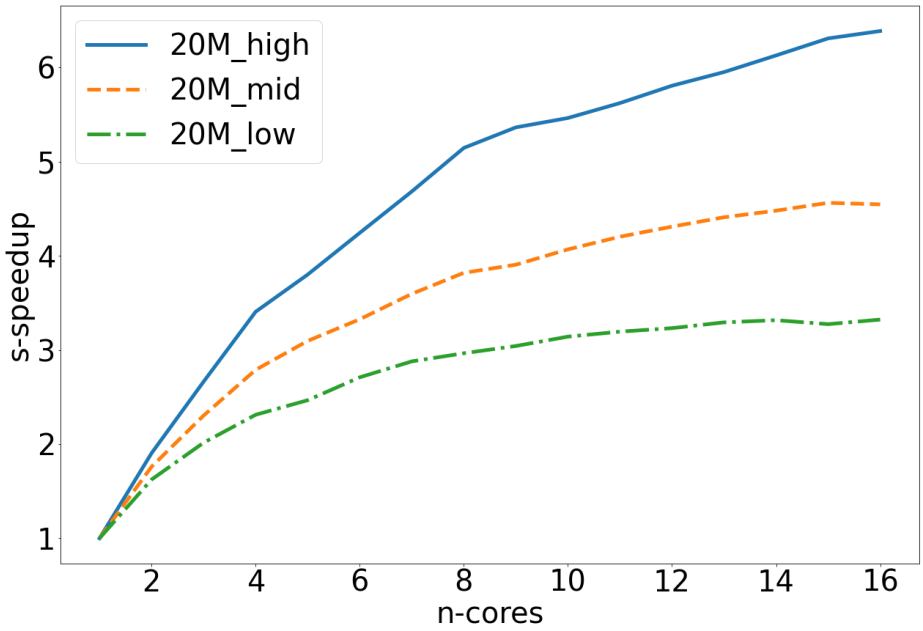


图 4 比较哈希分组在不同词频下加速比

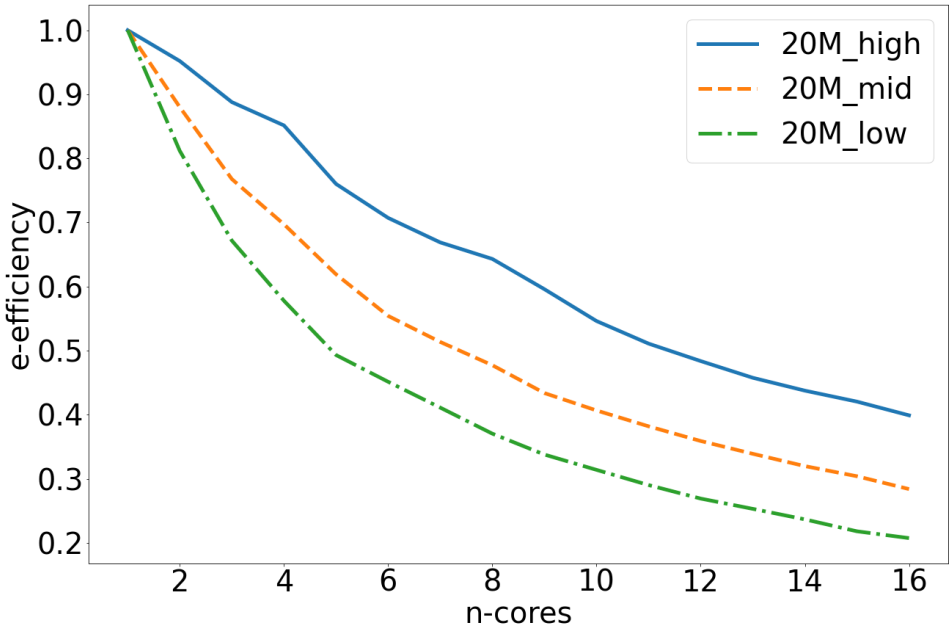
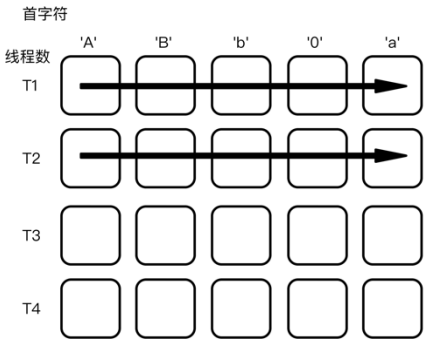


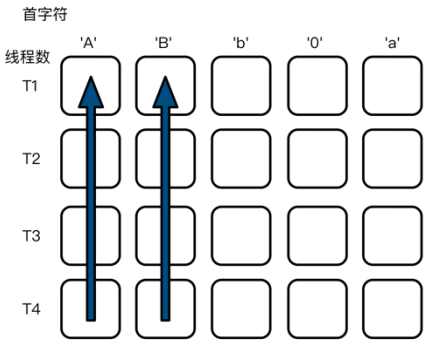
图 5 哈希方法在不同词频下的效率

在高词频情况下，哈希方法不重复储存相同的字符串，减小了内存开支，提高了缓存命中率和访存速度。实现了约低词频下两倍的加速比。

4.4 二维分组哈希

当前的哈希方法，仍然在合并多个线程的哈希表时存在大量串行部分，我们采取按操作线程和首字母分块的二维哈希方法。





如图所示，我们先直接将读入分成线程个数，每个线程内部对 62 种首字符情况开辟哈希表，按线程数哈希分组。然后我们对每个线程创建的某个首字符的哈希表进行并行的合并，最后实现哈希和合并都以并行处理。

测试环境	使用代码	数据集	方法
测试环境 2	代码段 4	20M_low.txt, 20M_mid.txt, 20M_high.txt	2D-hash

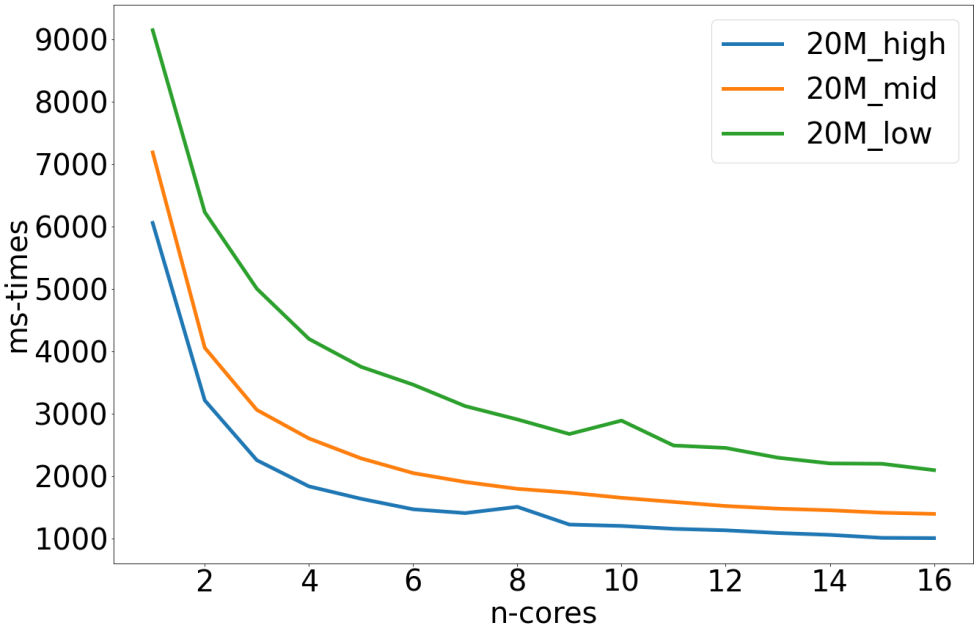


图 6 2D-hash 用时

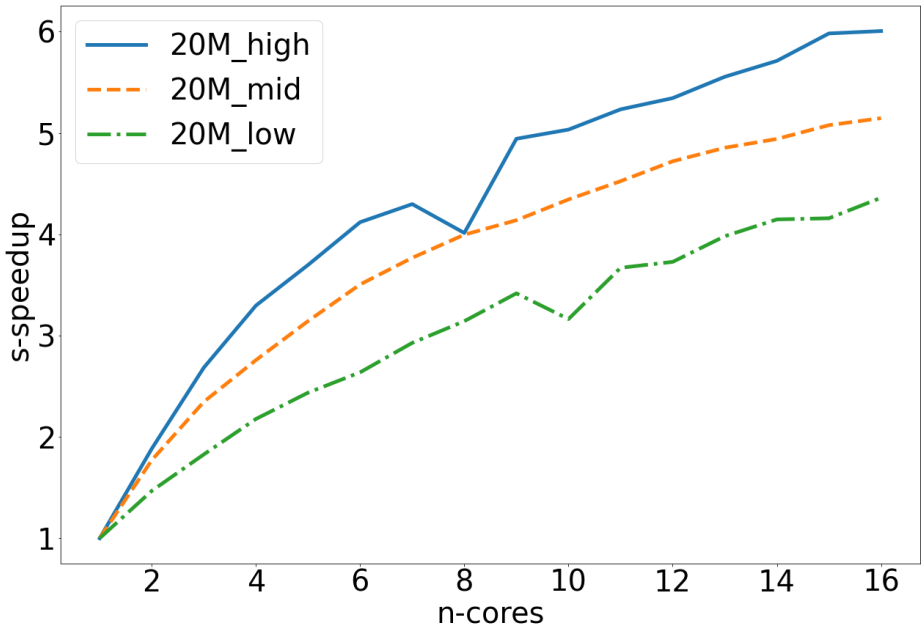


图 7 2D-hash 加速比

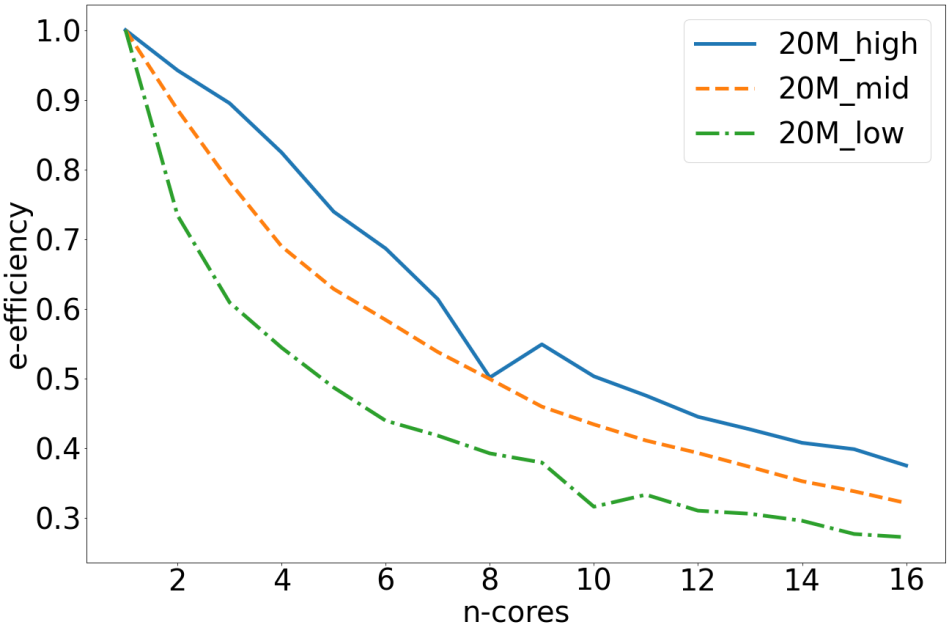


图 8 2D-hash 效率

在该方法下，并行效率进一步提高，低词频下加速比从原方法的 3.3 提升到 4.5。但是在高频 8 线程和低频 10 线程情况下均出现性能较大衰减，踩车是由于按该线程数分组后出现余数较大并且发生较多哈希碰撞的影响。但总体而言已经实现了较高的并行效率。

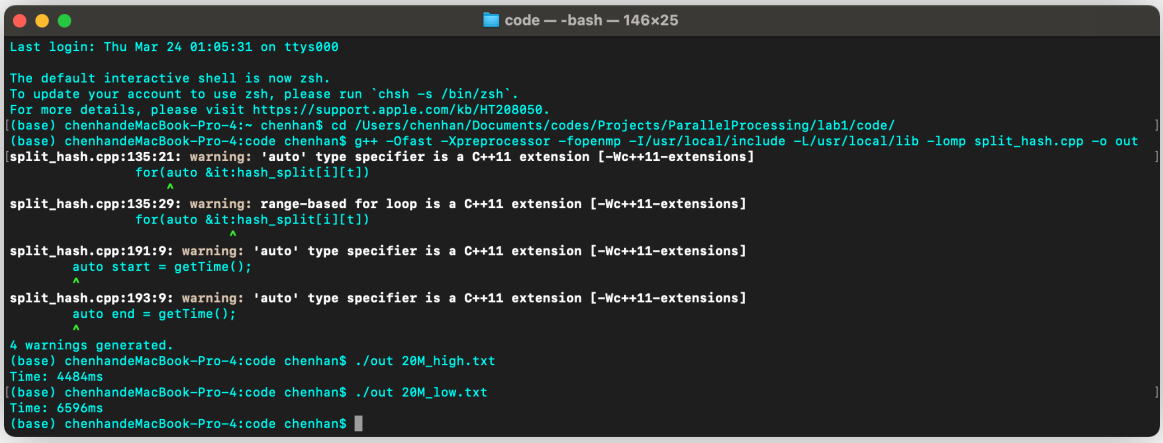


图 9 运行截图

5、总结

经过本次实验，我大致了解了 openmp 在并行处理中的使用方法，感受到了分组策略对并行处理效率提升的影响，同时比较了哈希和排序等算法在去重操作中的优缺点。

6、附件

./code:

mat_vect_mult.c	split.cpp	split_trie.cpp	trie.h
odd_even.c	split_hash.cpp	trap.c	trie.hpp

./image:

2D-hash.jpg	sort vs prefix sort.jpg
2D-hash.png	sort vs prefix sort.png
2D-hash.pxd	speedup (1).png
efficiency (1).png	speedup (2).png
efficiency (2).png	speedup (3).png
efficiency.png	speedup (4).png
hash speedup.png	speedup.png
hash t.png	time (1).png
sort vs prefix sort (1).png	time.png

./logs:

20M_High.xlsx	20M_low.csv	20_high.txt	20_low_2.txt
20M_Mid.xlsx	20M_mid.csv	20_high_2.txt	20_mid.txt
20M_high.csv	20_Low.xlsx	20_low.txt	20_mid_2.txt

./report:

《并行程序设计》实验报告-201936380086 陈涵.doc
《并行程序设计》实验报告一：OpenMP 编程.doc

./submit:

groupByString.cpp
《并行程序设计》实验报告-201936380086 陈涵.doc
《并行程序设计》实验报告-201936380086 陈涵.pdf