并行程序设计与算法实验 5

实验	OpenMP实现矩阵乘法	专业	计算机科学与技术
学号	21311525	姓名	马梓培
Email	mazp@mail2.sysu.edu.cn	完成日期	2024/04/22

1. 实验要求

1. OpenMP通用矩阵乘法
   - 使用OpenMP实现并行通用矩阵乘法
   - 设置线程数量（1-16）、矩阵规模（128-2048）、调度方式
     - 调度方式包括默认调度、静态调度、动态调度
   - 调度方式包括默认调度、静态调度、动态调度
2. 使用Pthreads构建并行for循环分解、分配、执行机制
   - 使用Pthreads构建并行for循环分解、分配、执行机制
   - 生成包含parallel_for函数的动态链接库（.so）文件

```
parallel_for(int start, int end, int inc,
             void *(*functor)(int,void*), void *arg, int num_threads)
```

start, end, inc分别为循环的开始、结束及索引自增量

- functor为函数指针，定义了每次循环所执行的内容
- arg为functor的参数指针，给出了functor执行所需的数据
- num_threads为期望产生的线程数量

2. OpenMP通用矩阵乘法

本实验代码见附件或github.

> OpenMP允许程序员只需要简单地申明一块代码应该并行执行，而由编译器和运行时系统来决定哪个线程具体执行哪个任务。

最基本的parallel指令可以以如下简单的形式来表示：

```
# pragma omp parallel
```

而在本实验中，我们通过如下parallel指令来实现并行：

```
# pragma omp parallel num_threads(num_thread)
```

这与基础的指令不同之处，在于多了个num_thread子句，从而可以允许程序员指定执行后代码块的线程数。

2.1 矩阵乘法

与之前几次实验类似，我们需要确定矩阵A需要运算的行，其基于线程的rank，可以通过omp_get_thread_num()函数来获得。

因此，矩阵的乘法如下所示：

```
void omp_matrix_mul() {
    int rank = omp_get_thread_num();
    int first_row = rank * avg_rows;
    int last_row = (rank + 1) * avg_rows;
    for (int i = first_row; i < last_row; i ++) {
        for (int j = 0; j < N; j ++) {
            float temp = 0;
            for (int z = 0; z < N; z ++) {
                temp += A[i * N + z] * B[z * N + j];
            }
            C[i * N + j] = temp;
        }
    }
}
```

将矩阵乘法函数结合parallel指令，从而实现并行化：

```
# pragma omp parallel num_threads(num_thread)
omp_matrix_mul();
```

2.2 程序计时

计时框架如下，所示，需要注意的是，可能usec不够减：

```
#include <sys/time.h>
...
struct timeval start_time, end_time;

gettimeofday(&start_time, NULL);
...
```

```
gettimeofday(&end_time, NULL);

long seconds = end_time.tv_sec - start_time.tv_sec;
long useconds = end_time.tv_usec - start_time.tv_usec;

// 如果结束时间的微秒数小于开始时间的微秒数，需要调整
if (useconds < 0) {
    useconds += 1000000;
    seconds -= 1;
}

double total_seconds = seconds + useconds / 1000000.0;
printf("Time Elapsed = %.6f seconds\n", total_seconds);
```

3. 实验结果

N = 128:

```
PP5 ) ./mm.out 1
Time Elapsed = 0.005874 seconds
PP5 ) ./mm.out 2
Time Elapsed = 0.003123 seconds
PP5 ) ./mm.out 4
Time Elapsed = 0.001781 seconds
PP5 ) ./mm.out 8
Time Elapsed = 0.003686 seconds
PP5 ) ./mm.out 16
Time Elapsed = 0.001638 seconds
```

N = 256:

```
PP5 ) ./mm.out 1
Time Elapsed = 0.047781 seconds
PP5 ) ./mm.out 2
Time Elapsed = 0.024062 seconds
PP5 ) ./mm.out 4
Time Elapsed = 0.012173 seconds
PP5 ) ./mm.out 8
Time Elapsed = 0.016812 seconds
PP5 ) ./mm.out 16
Time Elapsed = 0.009599 seconds
```

N = 512:

```
PP5 ) ./mm.out 1
Time Elapsed = 0.385720 seconds
PP5 ) ./mm.out 2
Time Elapsed = 0.196501 seconds
PP5 ) ./mm.out 4
Time Elapsed = 0.102242 seconds
PP5 ) ./mm.out 8
Time Elapsed = 0.063299 seconds
PP5 ) ./mm.out 16
Time Elapsed = 0.066805 seconds
```

N = 1024:

```
PP5 ) ./mm.out 1
Time Elapsed = 3.464608 seconds
PP5 ) ./mm.out 2
Time Elapsed = 1.738003 seconds
PP5 ) ./mm.out 4
Time Elapsed = 0.874890 seconds
PP5 ) ./mm.out 8
Time Elapsed = 0.533318 seconds
PP5 ) ./mm.out 16
Time Elapsed = 0.521655 seconds
```

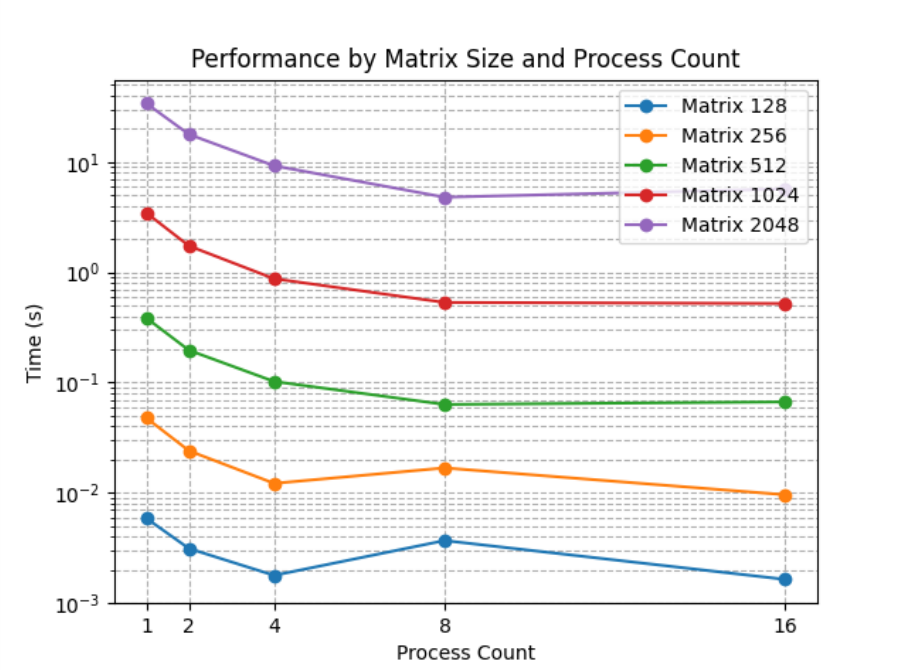
N = 2048:

```
PP5 ) ./mm.out 1
Time Elapsed = 34.086398 seconds
PP5 ) ./mm.out 2
Time Elapsed = 17.859157 seconds
PP5 ) ./mm.out 4
Time Elapsed = 9.262058 seconds
PP5 ) ./mm.out 8
Time Elapsed = 4.804137 seconds
PP5 ) ./mm.out 16
Time Elapsed = 5.713462 seconds
```

表格：

P\N	128	256	512	1024	2048
1	0.005874	0.047781	0.385720	3.464608	34.086398
2	0.003123	0.024062	0.196501	1.738003	17.859157
4	0.001781	0.012173	0.102242	0.874890	9.262058
8	0.003686	0.016812	0.063299	0.533318	4.804137
16	0.001638	0.009599	0.066805	0.521655	5.713462

可视化：



在使用OpenMP进行矩阵乘法的实验中，我们观察到随着矩阵规模的增加，运行时间显著增长。增加进程数可以有效缩短运算时间，特别是当进程数不超过机器的物理核心数（如8核）时，性能提升最为明显。例如，对于2048规模的矩阵，使用8个进程比使用16个进程具有更低的运行时间，显示出并行化的边界效益。这些观察结果表明，在应用并行计算时，选择适宜的进程数是至关重要的，以确保计算资源的高效利用，同时避免因超出物理核心数，由于进程上下文切换而引入的额外并行开销。

4. 不同调度方式实现

不同调度方式，如静态和动态调度，可以通过预处理指令来实现：

```
1. #pragma omp parallel for schedule(static, 1) num_threads(num_thread)
2. #pragma omp parallel for schedule (dynamic, 1) num_threads(num_thread)
```

4.1 静态调度

N = 128:

```
PP5 ) ./mm.out 1
Time Elapsed = 0.005855 seconds
PP5 ) ./mm.out 2
Time Elapsed = 0.003015 seconds
PP5 ) ./mm.out 4
Time Elapsed = 0.001751 seconds
PP5 ) ./mm.out 8
Time Elapsed = 0.014824 seconds
PP5 ) ./mm.out 16
Time Elapsed = 0.001821 seconds
```

N = 256:

```
PP5 ) ./mm.out 1
Time Elapsed = 0.048124 seconds
PP5 ) ./mm.out 2
Time Elapsed = 0.024265 seconds
PP5 ) ./mm.out 4
Time Elapsed = 0.012359 seconds
PP5 ) ./mm.out 8
Time Elapsed = 0.015369 seconds
PP5 ) ./mm.out 16
Time Elapsed = 0.008438 seconds
```

N = 512:

```
PP5 ) ./mm.out 1
Time Elapsed = 0.404326 seconds
PP5 ) ./mm.out 2
Time Elapsed = 0.201948 seconds
PP5 ) ./mm.out 4
Time Elapsed = 0.100092 seconds
PP5 ) ./mm.out 8
Time Elapsed = 0.072592 seconds
PP5 ) ./mm.out 16
Time Elapsed = 0.064959 seconds
```

N = 1024:

```
PP5 ) ./mm.out 1
Time Elapsed = 3.726951 seconds
PP5 ) ./mm.out 2
Time Elapsed = 1.821126 seconds
PP5 ) ./mm.out 4
Time Elapsed = 0.931003 seconds
PP5 ) ./mm.out 8
Time Elapsed = 0.620243 seconds
PP5 ) ./mm.out 16
Time Elapsed = 0.576337 seconds
```

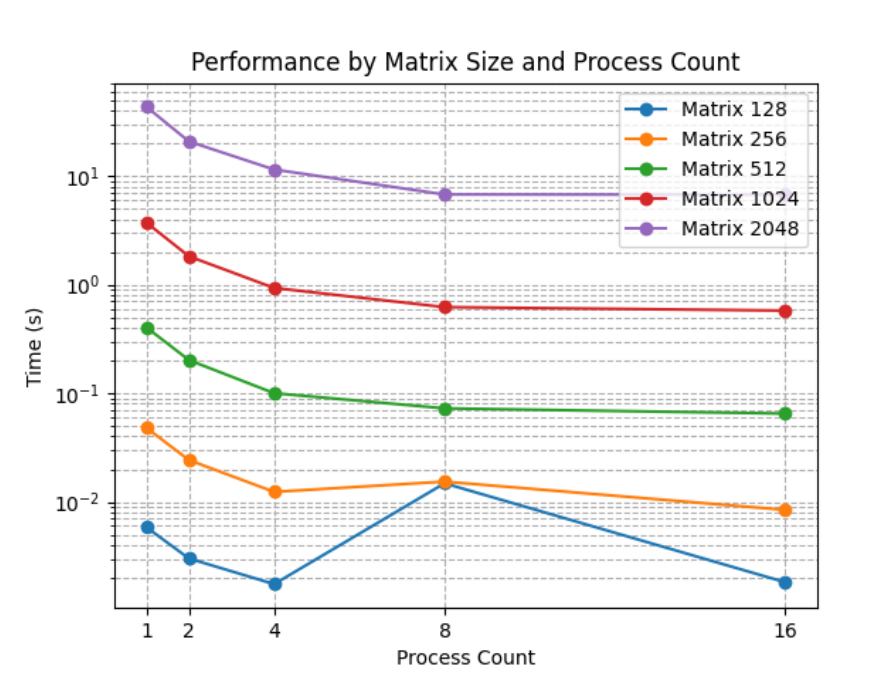
N = 2048:

```
PP5 ) ./mm.out 1
Time Elapsed = 43.360471 seconds
PP5 ) ./mm.out 2
Time Elapsed = 20.762046 seconds
PP5 ) ./mm.out 4
Time Elapsed = 11.449756 seconds
PP5 ) ./mm.out 8
Time Elapsed = 6.752237 seconds
PP5 ) ./mm.out 16
Time Elapsed = 6.712368 seconds
```

表格：

P\N	128	256	512	1024	2048
1	0.005855	0.048124	0.404326	3.726951	43.360471
2	0.003015	0.024265	0.201948	1.821126	20.762046
4	0.001751	0.012359	0.100092	0.931003	11.449756
8	0.014824	0.015369	0.072592	0.620243	6.752237
16	0.001821	0.008438	0.064959	0.576337	6.712368

可视化：



在使用静态调度方式进行OpenMP矩阵乘法实验时，我们同样观察到随着矩阵规模的增大，运行时间增长。与默认调度相比，静态调度效率降低。例如，对于2048规模的矩阵，静态调度在使用8核和16核时的运行时间高于默认调度，表明默认调度在任务分配上可能更有效率。

4.2 动态调度

N = 128:

```
PP5 ) ./mm.out 1
Time Elapsed = 0.005983 seconds
PP5 ) ./mm.out 2
Time Elapsed = 0.003115 seconds
PP5 ) ./mm.out 4
Time Elapsed = 0.001711 seconds
PP5 ) ./mm.out 8
Time Elapsed = 0.001057 seconds
PP5 ) ./mm.out 16
Time Elapsed = 0.001329 seconds
```

N = 256:

```
PP5 ) ./mm.out 1
Time Elapsed = 0.048604 seconds
PP5 ) ./mm.out 2
Time Elapsed = 0.024398 seconds
PP5 ) ./mm.out 4
Time Elapsed = 0.012336 seconds
PP5 ) ./mm.out 8
Time Elapsed = 0.012760 seconds
PP5 ) ./mm.out 16
Time Elapsed = 0.012713 seconds
```

N = 512:

```
PP5 ) ./mm.out 1
Time Elapsed = 0.398889 seconds
PP5 ) ./mm.out 2
Time Elapsed = 0.200865 seconds
PP5 ) ./mm.out 4
Time Elapsed = 0.100701 seconds
PP5 ) ./mm.out 8
Time Elapsed = 0.057415 seconds
PP5 ) ./mm.out 16
Time Elapsed = 0.055641 seconds
```

N = 1024:

```
PP5 ) ./mm.out 1
Time Elapsed = 3.642851 seconds
PP5 ) ./mm.out 2
Time Elapsed = 1.848243 seconds
PP5 ) ./mm.out 4
Time Elapsed = 0.944562 seconds
PP5 ) ./mm.out 8
Time Elapsed = 0.547553 seconds
PP5 ) ./mm.out 16
Time Elapsed = 0.548499 seconds
```

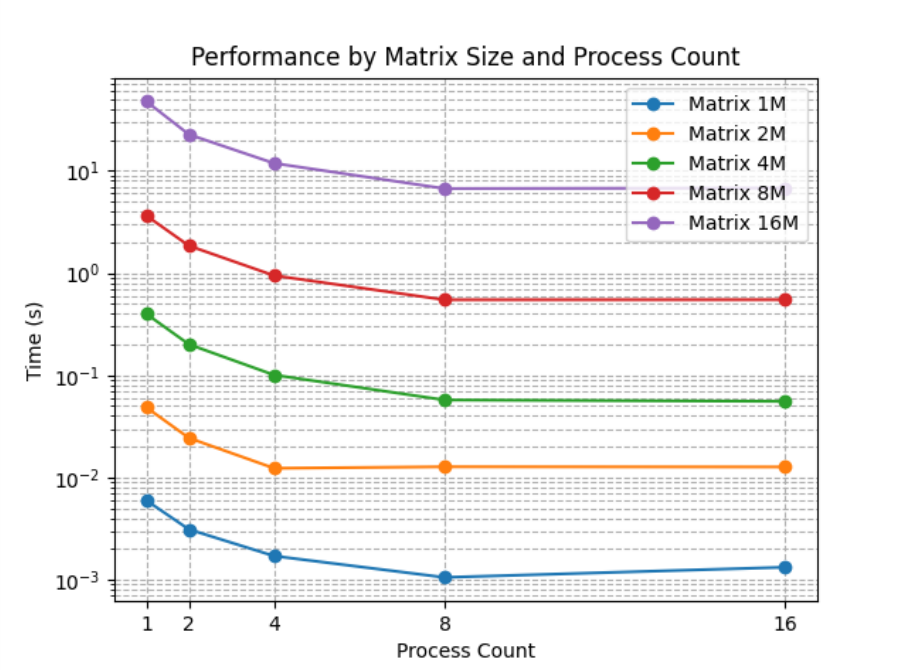
N = 2048:

```
PP5 ) ./mm.out 1
Time Elapsed = 47.677945 seconds
PP5 ) ./mm.out 2
Time Elapsed = 22.641008 seconds
PP5 ) ./mm.out 4
Time Elapsed = 11.833128 seconds
PP5 ) ./mm.out 8
Time Elapsed = 6.701250 seconds
PP5 ) ./mm.out 16
Time Elapsed = 6.779896 seconds
```

表格：

P\N	128	256	512	1024	2048
1	0.005983	0.048604	0.398889	3.642851	47.677945
2	0.003115	0.024398	0.200865	1.848243	22.641008
4	0.001711	0.012336	0.100701	0.944562	11.833128
8	0.001057	0.012760	0.057415	0.547553	6.701250
16	0.001329	0.012713	0.055641	0.548499	6.779896

可视化：



在使用动态调度方式实现OpenMP矩阵乘法时，我们观察到随着进程数的增加，性能下降，尤其是在较大矩阵规模（如2048）上，动态调度的性能劣于默认和静态调度。可能是因为核的工作量基本相同，而动态调度更加适合核的工作量不同的情况，所以可能动态调动对于性能的提升不高，甚至在本实验中导致了性能下降。

5. Pthreads构建并行for循环

5.1 构建parallel_for函数

```
1  void parallel_for(int start, int end, int inc, void *(*functor)(int, void*), void
   *arg, int num_threads) {
2      pthread_t threads[num_threads];
3      ThreadData thread_data[num_threads];
4      int chunk_size = (end - start) / num_threads;
5
6      for (int i = 0; i < num_threads; i++) {
7          thread_data[i].start = start + i * chunk_size;
8          thread_data[i].end = (i == num_threads - 1) ? end : thread_data[i].start +
   chunk_size;
9          thread_data[i].inc = inc;
10         thread_data[i].functor = functor;
11         thread_data[i].arg = arg;
12         pthread_create(&threads[i], NULL, thread_function, &thread_data[i]);
13     }
14
15     for (int i = 0; i < num_threads; i++) {
16         pthread_join(threads[i], NULL);
17     }
18 }
```

利用多线程技术将一个较大的任务分割成多个小块，让每个线程并行处理一部分任务，从而提高程序的执行效率。

编译为.so文件：

```
gcc -shared -fpic -o libparallelfor.so parallel_for.c
```

5.2 引用parallel_for函数

```
// 需要extern声明以便链接到动态库
extern void parallel_for(int start, int end, int inc, void *(*func)(int, void*), void *arg, int num_threads);

...

parallel_for(0, N, 1, matrix_multiply_functor, &data, num_threads);
```

5.3 实验结果

编译mm_parallelfor.c

```
gcc -o matrix_mult.out mm_parallelfor.c -L. -lparallelfor -lpthread -fopenmp
```

将当前目录加入到环境变量：

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
```

N = 128:

```
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 1
Total time: 0.006226 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 2
Total time: 0.003153 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 4
Total time: 0.001921 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 8
Total time: 0.001709 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 16
Total time: 0.001755 seconds
```

N = 256:

```
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 1
Total time: 0.048846 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 2
Total time: 0.024648 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 4
Total time: 0.012420 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 8
Total time: 0.019075 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 16
Total time: 0.009580 seconds
```

N = 512:

```
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 1
Total time: 0.419104 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 2
Total time: 0.210839 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 4
Total time: 0.113997 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 8
Total time: 0.083029 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 16
Total time: 0.071260 seconds
```

N = 1024:

```
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 1
Total time: 4.707591 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 2
Total time: 2.551885 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 4
Total time: 1.340333 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 8
Total time: 0.768914 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 16
Total time: 0.815397 seconds
```

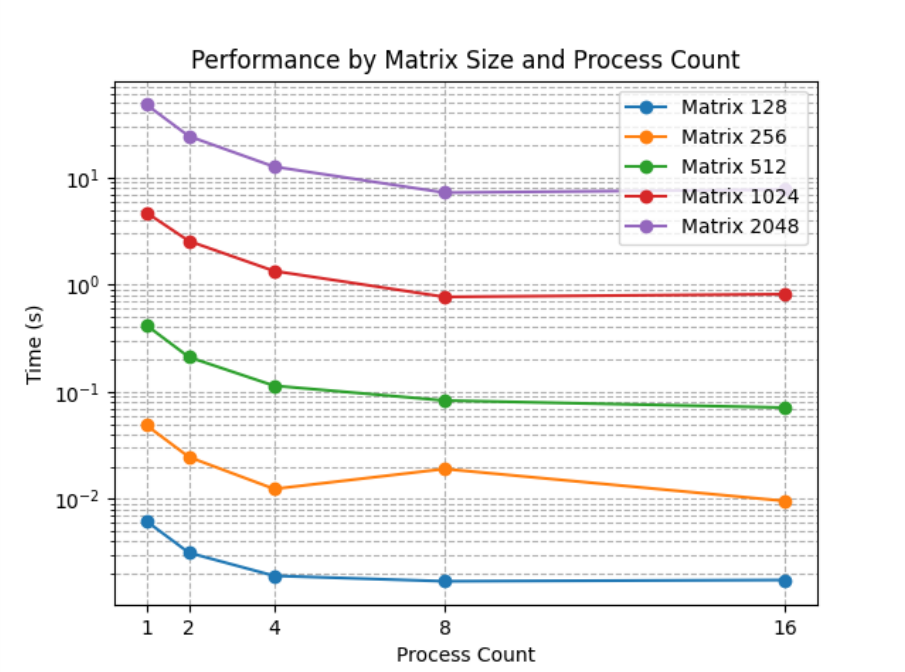
N = 2048:

```
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 1
Total time: 47.903420 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 2
Total time: 24.262633 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 4
Total time: 12.605440 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 8
Total time: 7.224184 seconds
xiaoma@xiaoma-virtual-machine:~/Parallel-Programming/PP5$ ./matrix_mult.out 16
Total time: 7.712825 seconds
```

表格：

P\N	128	256	512	1024	2048
1	0.006226	0.048846	0.419104	4.707591	47.903420
2	0.003153	0.024648	0.210839	2.551885	24.262633
4	0.001921	0.012420	0.113997	1.340333	12.605440
8	0.001709	0.019075	0.083029	0.768914	7.224184
16	0.001755	0.009580	0.071260	0.815397	7.712825

可视化：



what a pity，似乎效率还不如前面openMP的预处理指令。。

6. 实验感想

通过这次的并行程序设计与算法实验，我对并行编程有了更深入的理解和体验。在本次实验中，我使用了OpenMP和Pthreads两种不同的技术来实现矩阵乘法的并行化。实验过程中，我深刻体会到了并行编程在处理大规模计算任务时的强大效能。

首先，通过OpenMP实现的矩阵乘法让我感受到了并行化带来的直观性能提升。尤其是在调整线程数量和选择不同的调度方式时，我可以明显看到运行时间的变化。这不仅验证了并行计算的有效性，也让我认识到在实际应用中选择合适的线程数量和调度策略的重要性。

然而，通过Pthreads手动构建并行for循环的过程虽然较为复杂，却提供了更多的控制空间。尽管最终的性能可能不如使用OpenMP的简便方法，这一过程却加深了我对线程管理和资源分配的理解。

此外，实验中的性能测试和数据可视化也让我认识到了评估并行程序性能的重要性。通过对比不同配置下的执行时间，我更加明白了如何根据具体任务和硬件条件来优化并行程序。