

# 目 录

<b>1</b>	<b>目标.....</b>	<b>1</b>
<b>2</b>	<b>假设.....</b>	<b>2</b>
2.1	串行算法复杂度 .....	2
2.2	并行实现正确性 .....	3
2.3	大数场景分析 .....	4
2.4	并行优化方案设计 .....	5
<b>3</b>	<b>方法.....</b>	<b>6</b>
<b>4</b>	<b>结果.....</b>	<b>7</b>
<b>5</b>	<b>结论.....</b>	<b>11</b>

# 1 目标

- 分析串行算法复杂度
  - 时间复杂度
  - 空间复杂度
- 分析并行实现的正确性
  - 可行算法的描述分析
  - 工作分解
  - 选择编程模型
- 大数场景分析
- 并行优化方案设计

## 2 假设

### 2.1 串行算法复杂度

由于需要与后面串行算法进行比较，因此在这里就使用通项公式进行计算斐波那契数列。

$$a_n = \frac{1}{\sqrt{5}} \left[ \left( \frac{1+\sqrt{5}}{2} \right)^n - \left( \frac{1-\sqrt{5}}{2} \right)^n \right]$$

这样就不难想到实现该代码的方法了。

主要测试代码：

```
1. for(i = 2; i <= n; ++i){  
2.     fiboArray[i] = fibo(i);  
3. }
```

#### 1. 时间复杂度分析：

由于只有一层代码，因此时间复杂度分析起来比较简单。利用通项公式计算斐波那契数列中 `pow` 在 STL 里时间复杂度为  $O(1)$ ，那么显然串行斐波那契时间复杂度为  $O(n)$ 。

#### 2. 空间复杂度分析：

空间复杂度也比较好分析，为  $O(n)$ 。

## 2.2 并行实现正确性

本机上没有N V I D I A显卡，因此做不了C U D A分析，所以在这里就讲一讲三种并行的情况。大部分具体实现体现在实验报告中，而不是本思考题中。因此本文主要是比较串并行以及分析。

### P t h r e a d

这种方法是比较常见的实现并行的方法，也是用的比较多的方法。但是在这里p t h r e a d的效果可能不是很好。

第一因为斐波那契计算的工作量比较小，如果给每个计算任务都分配线程，那么创建开销会极大，由此带来的后果就是效率极低，创建开销远远超过多线程计算带来的收益。

第二如果按照线程数来分配每个线程的任务的话，又会带来计算上的额外开销，这种开销就是为每个线程分配任务的起始位置以及结束位置，如果线程任务分配的不是很好的话，会大大影响效率。

因此这种方案最后得到的效果很可能与理想中差距甚远。

### O p e n M P

O p e n M P是一种已经比较成熟的方案了，因此在正确性上没有什么问题。框架的话也比较简单，在f o r循环前加入规定语句即可，但是要注意使用条件。具体实施已经在实验报告中阐述的十分清楚了，在此不赘述。

理论上，这种情况一定会带来效率上的提升。

### M P I

M P I也是一种并行编程方案，是一个跨语言的通讯协议。与OpenMP并行程序不同，MPI是一种基于信息传递的并行编程技术。将参数传给其他进程，就可以在多个进程里共同计算整个斐波那契数列的值了。在主进程里只需要接受传过来的数据即可。从理论角度来说，M P I函数都是封装好的，我们只需要调用，同时，传入参数也不像p t h r e a d那么复杂，只需要根据每个进程的进程号就可以划分相应的工作范围。因此，并行加速的概率会很大。

## 2.3 大数场景分析

由于在实验中存储斐波那契数列值用的是 `long long` 类型，所以最大能计算到第 93 项。

对于大数情况，猜想使用并行计算效率会高效得多，因为随着计算的工作量增大，大多数时间会消耗在这个上面，而不是为了并行而产生的其它开销上。因此在这种情况下计算加速比是十分合理的。

对于 `pthread`，并行加速效果可能不会很好，对于 `OpenMP`，并行的效果会有，但是可能不是很明显。对于 `MPI`，由于本机 2 核，因此使用 4 线程来进行计算任务，初步推测理想加速比为 3（因为有一个进程需要用来输出）。

## 2.4 并行优化方案设计

对于M P I来说，在代码上可能没有什么优化的地方了，因此如果想要获得更大的加速比，就需要从理论上突破，如增加进程数。但是由于4个进程已经是一个上限了，因此在这里就做一个2，3，4的比较。来体现优化效果。

同理，对于O p e n M P，在这里就更改变参与f o r循环的t h r e a d数就行了。

但是如果进一步提高线程数，可能优化的情况就不会十分明显，很有可能到后来随着线程数的增加，时间也几乎不会减少了。

### 3 方法

本次主要就是计算加速比，因此首先就要得到原始串行算法的运行时间，因此首先测试串行算法计算斐波那契数列的值（使用通项）。

然后对于 `p t h r e a d` 并行编程方法，使用同样的方法进行测试。

对于 `O p e n M P`，可以调节线程数来测试不同的情况，改变 `num_threads()` 中的值即可。分别记录它们的运行时间。

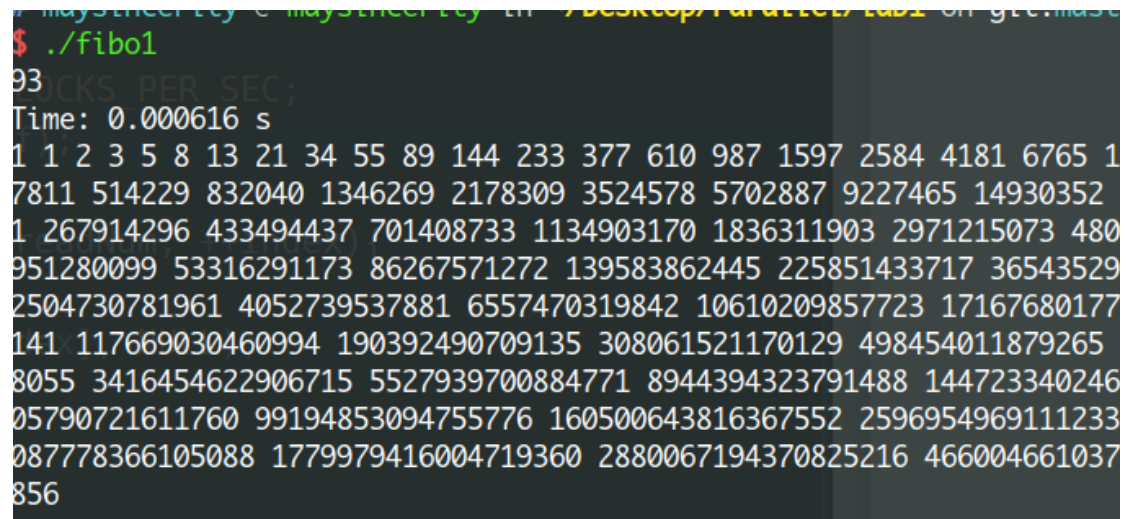
对于 `M P I`，同样的，需要改变运行进程的数量来分析不同情况下的并行优化效果。

需要注意的是，这里的时间并不是所有的时间，在这里只考虑了计算出斐波那契数列时间，因为后面输出斐波那契数列是串行的，没有计算时间的必要，也不能进行。

## 4 结果

### 1. 串行算法时间

首先需要计算串行斐波那契数列到底花了多久，因此我在本机上测试了多次，最后发现平均时间为  $0.0006\text{ s} - 0.0007\text{ s}$  左右。



```
$ ./fibo1
93
Time: 0.000616 s
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 1
7811 514229 832040 1346269 2178309 3524578 5702887 9227465 14930352
1 267914296 433494437 701408733 1134903170 1836311903 2971215073 480
951280099 53316291173 86267571272 139583862445 225851433717 36543529
2504730781961 4052739537881 6557470319842 10610209857723 17167680177
141 117669030460994 190392490709135 308061521170129 498454011879265
8055 3416454622906715 5527939700884771 8944394323791488 144723340246
05790721611760 99194853094755776 160500643816367552 2596954969111233
087778366105088 1779979416004719360 2880067194370825216 466004661037
856
```

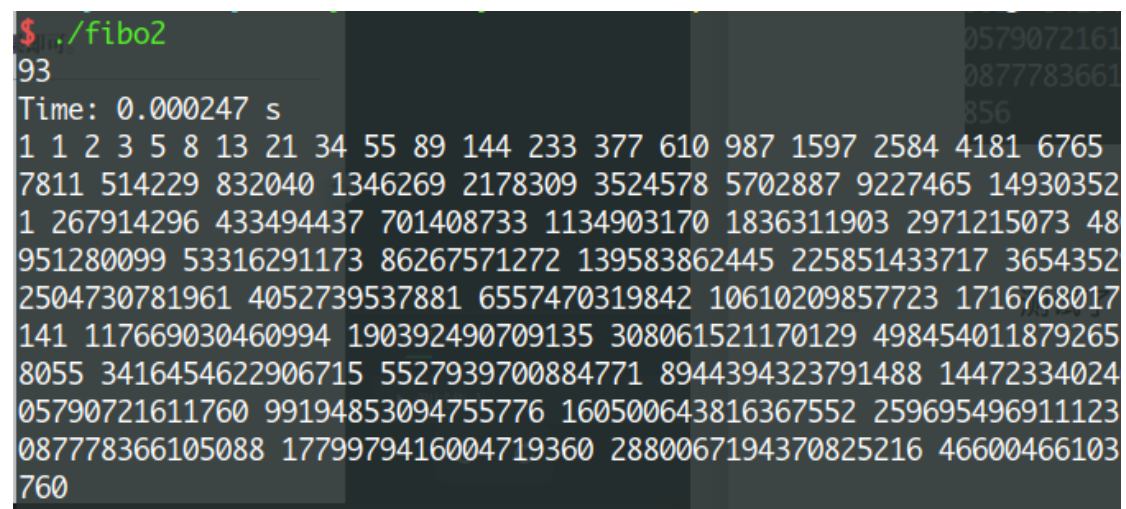
图 4-1 串行时间

测试了 10 次，最后得出总时间为  $0.006502$

### 2. OpenMP 算法时间

之后计算使用 OpenMP 算法的并行程序时间。

使用 3 个线程，发现平均时间都在  $0.000240\text{ s}$  左右。



```
$ ./fibo2
93
Time: 0.000247 s
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
7811 514229 832040 1346269 2178309 3524578 5702887 9227465 14930352
1 267914296 433494437 701408733 1134903170 1836311903 2971215073 48
951280099 53316291173 86267571272 139583862445 225851433717 3654352
2504730781961 4052739537881 6557470319842 10610209857723 1716768017
141 117669030460994 190392490709135 308061521170129 498454011879265
8055 3416454622906715 5527939700884771 8944394323791488 14472334024
05790721611760 99194853094755776 160500643816367552 259695496911123
087778366105088 1779979416004719360 2880067194370825216 46600466103
760
```

图 4-2 OpenMP 3 线程

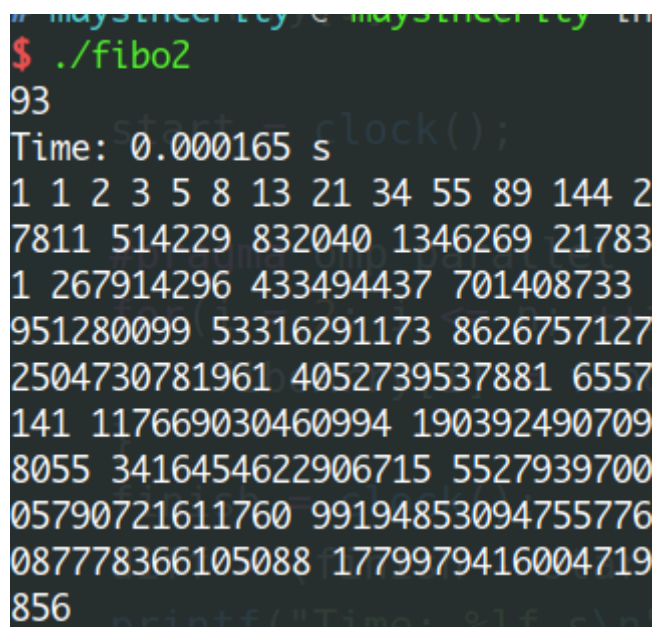


测试 10 次，得到总时间为 0.002363。

使用 3 个线程，理论上加速比应该为 3，但由于一些不可控因素，达到这个理想加速比几乎是不可能的。

经计算，加速比为 2.75，比较接近 3。如果数据集更大的话，应该会更加接近理想加速比。

对此进行优化，使用 4 个线程，发现平均时间都在 0.000170 s 左右。



```
# maystherley & maystherley th
$ ./fiboz
93
Time: 0.000165 s
1 1 2 3 5 8 13 21 34 55 89 144 2
7811 514229 832040 1346269 21783
1 267914296 433494437 701408733
951280099 53316291173 8626757127
2504730781961 4052739537881 6557
141 117669030460994 190392490709
8055 3416454622906715 5527939700
05790721611760 99194853094755776
087778366105088 1779979416004719
856
```

图 4-3 OpenMP 4 线程

测试 10 次，得到总时间为 0.001682。

经计算，得到加速比为 3.86。接近理想加速比 4。

### 3. P t h r e a d 算法时间

需要计算运行时间，在代码里设置的最大线程是 4，当数据量大的时候会分为 4 个线程进行计算。发现平均时间在 0.000190 s 左右。

```

$ ./fibo3
93
Time: 0.000185 s
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
7811 514229 832040 1346269 2178309 3524578
1 267914296 433494437 701408733 1134903170
951280099 53316291173 86267571272 1395838624
2504730781961 4052739537881 6557470319842
141 117669030460994 190392490709135 308061501
8055 3416454622906715 5527939700884771 8945354
05790721611760 99194853094755776 1605006437194
087778366105088 1779979416004719360 288006719
760

```

图 4-4 pthread 时间

测试 10 次，得到总时间 0.001867 s。

经计算，得到加速比 3.48。就结果来说，是远没有达到理想加速比 4 的。

#### 4. MPI 算法时间

需要计算运行时间，在运行时设置进程数为 4。发现平均时间在 0.000230 s 左右。

由于本机上只能设置进程数为 4，因此优化效果只能在 OpenMP 里面看了。

```

$ mpiexec -n 4 ./fibo4
93
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
7811 514229 832040 1346269 2178309 3524578
1 267914296 433494437 701408733 1134903170
951280099 53316291173 86267571272 1395838624
2504730781961 4052739537881 6557470319842
141 117669030460994 190392490709135 308061501
8055 3416454622906715 5527939700884771 8945354
05790721611760 99194853094755776 1605006437194
087778366105088 1779979416004719360 288006719
760
Time: 0.000229 s

```

图 4-5 MPI 时间

测试 10 次，得到总时间 0.002317 s。

经计算，得到加速比 2.81。与理想加速比 3 相比还是差了一点。

## 5 结论

本次实验串行复杂度并不高，为 $O(n)$ 。一般情况下不需要串行来提高效率。

然后就是3种并行方案了，分别是`pthread`，`OpenMP`，`MP I`。对于`pthread`，其实优化的不是很好，并且如果要使用线程，需要考虑的东西比其他的方式要稍微多一点。比如需要将计算范围作为参数传递，需要等待全部线程结束等。因此人为的优化效果可能没有像`OpenMP`这样封装好的工具高。在本次试验中，`OpenMP`的优化效果也是很好的，根据线程数的不同，优化效果略有区别，比如4线程的优化效果比3线程优化效果好，但是到了4线程之后更多线程的情况，可能时间上都差不多，没有效率。而`MP I`的优化效率应该也是可以接受的，从测试结果来看，`MP I`优化效率达到了2.81，作为横向比较的是`OpenMP`的2.75，可以看到两者优化效率不相上下，从这一测试集来看，`MP I`略优。

同时，在大数场景下并行的优化效率是要更好地，因为计算的比例增加了，在一定程度上抵消了并行的开销。

在并行优化这一块，如果从并行上入手，就可以考虑不同的优化方案，并且测试不同的并行线程/进程下的效率。如果从代码上入手的话，就要考虑怎么修改代码才能使其更适合并行，作业指导给的是通项公式，这是比较适合的，因为每个工作都不是相关的。如果用迭代法之类的，可能与之前的计算结果比较相关，并行性可能就不是那么好。