浙江大學

本科生课程实验报告

OpenMP PI

课程名称:	并行计算与多核编程
姓名:	陈卓
学院:	计算机科学与技术
专业:	计算机科学与技术
学号:	3170101214
指导老师:	楼学庆

2020 年 5 月 29 日

1 介绍

计算无理数 π 的值在数值计算中是一个长久的问题。自 Chunovsky 兄弟将 π 计算至 10 亿位后，Chunovsky 算法甚至成为了一种业余地测试计算机性能的方式。本次试验选取一种较为简单的算法和 Chunovsky 算法计算 π，并用 OpenMP 实现它们的并行算法。完整代码在 GitHub。

2 实验平台

(1) 2.3 GHz 双核 Intel Core i5

(2) clang 10.0.0

(3) OpenMP 8.0.0

3 算法

3.1 积分

我们可以通过简单的定积分来完成 π 的计算。

$$\pi = \int_0^1 \frac{4}{1+x^2} dx \tag{1}$$

由公式 1 我们可以获取一种计算 π 的方法，即将区间 $[0,1]$ 分为 N 份，通过累加的方式计算这个定积分。

代码如下。

```
std::string pi_simple(int N, int n_threads) {
   mpf_class sum = 0;
   mpf_class step = 1.0 / N;
   mpf_class x;
   for (int i = 0; i < N; i++) {
      x = ( i + 0.5 ) * step;
      sum += 4.0 / (1.0 + x * x);
   }
   mpf_class pi = sum*step;
   mp_exp_t exp;
   return pi.get_str(exp);
}
```

3.2 Chunovsky 算法

算法一简单易懂，但它的收敛速度自然是很慢的，N 扩大十倍大约能增加 2 位精度。为了提升收敛，众多数学家提出了各种各样的级数，Chunovsky 级数就是其中最有代表性之一。[1]

$$\frac{1}{\pi} = 12 \cdot \sum_{n=0}^{\infty} \frac{(-1)^n (6n)!}{(3n)!(n!)^3} \cdot \frac{13591409 + 545140134n}{640320^{3n+3/2}} \tag{2}$$

观察公式 2，其中每一项有大量的乘除法运算，而众所周知，乘除法的效率远比加减法低，因此可以用到 Binary Splitting 技术对计算过程加速。它的核心思想是将每一项的分数合并为加法，最终只进行一次或少量乘除法。[1][2]

代码如下。

```cpp
void bs(long a, long b, mpz_class &P, mpz_class &Q, mpz_class &T) {
    if (b - a == 1) {
        if (a == 0) {
            P = Q = mpz_class(1);
        } else {
            P = mpz_class((6*a-5)*(2*a-1)*(6*a-1));
            Q = mpz_class(a*a*a*C3_OVER_24);
        }
        T = P * (13591409 + 545140134*a);
        if (a%2 == 1) {
            T = -T;
        }
    } else {
        long m = (a + b) / 2;
        mpz_class Pam, Qam, Tam;
        mpz_class Pmb, Qmb, Tmb;
        bs(a, m, Pam, Qam, Tam);
        bs(m, b, Pmb, Qmb, Tmb);
        P = Pam * Pmb;
        Q = Qam * Qmb;
        T = Qmb * Tam + Pam * Tmb;
    }
}

std::string pi_chudnovsky(unsigned int digits) {
    mpf_set_default_prec(digits*log2(10));
    mpf_class DIGITS_PER_TERM = log(C3_OVER_24.get_d()/6/2/6);
    mpf_class a = digits/DIGITS_PER_TERM + 1;
    long N = a.get_si();
    mpz_class P, Q, T;
    bs(0, N * 10, P, Q, T);
    mpf_class pi = (Q * 426880 * my_sqrt(10005)) / T;
```

[1] 因为这次实验侧重点在计算，Chunovsky 级数的证明在此不做叙述。

```
    mp_exp_t exp;
    std::string ret = pi.get_str(exp);
    return ret;
}
```

4 用 OpenMP 实现的并行算法

对于算法一，我们可以将 OpenMP 运用在循环中，并用 reduction 处理 `sum` 变量。相关代码如下。

```
std::string pi_simple(int N, int n_threads) {
    ...
    #pragma omp parallel for private( x ) reduction(+: sum) num_threads(n_threads)
    for (int i = 0; i < N; i++) {
        x = ( i + 0.5 ) * step;
        sum += 4.0 / (1.0 + x * x);
    }
    ...
}
```

对于算法二，我们可以将 OpenMP 运用在 `bs` 函数的递归调用中，因此需要一个变量记录递归深度，如果线程用完则串行执行。相关代码如下。

```
void bs(long a, long b, mpz_class &P, mpz_class &Q, mpz_class &T, int max_depth=0)
{
    ...
    if (max_depth > 0) {
    #pragma omp parallel sections default(shared)
    {
    #pragma omp section
        {
            bs(a, m, Pam, Qam, Tam, max_depth - 1);
        }
    #pragma omp section
        {
            bs(m, b, Pmb, Qmb, Tmb, max_depth-1);
        }
    }
    } else {
        bs(a, m, Pam, Qam, Tam, max_depth);
        bs(m, b, Pmb, Qmb, Tmb, max_depth);
    }
    ...
}
```

5 测试结果与讨论

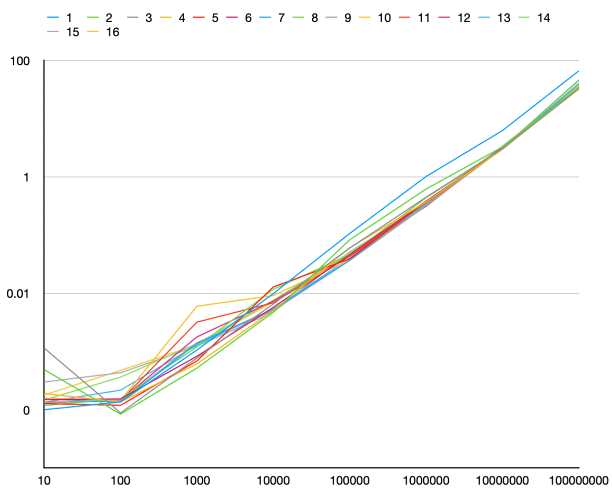
对于算法一，因为它的收敛性达不到要求，因此我们只关注 OpenMP 的加速效果。

	10	100	1000	10000	100000	1000000	10000000	100000000
1	0.0000998974	0.000136137	0.00106001	0.00989914	0.106428	1.01387	6.24572	66.3055
2	0.000487089	0.0000839233	0.000519037	0.00472999	0.0830519	0.617272	3.26332	34.5744
3	0.00114298	0.0000879765	0.000777006	0.00655293	0.0597801	0.447235	3.06532	32.6925
4	0.000189066	0.000135899	0.000613928	0.00505185	0.060715	0.370304	3.0075	32.7761
5	0.000127077	0.000118971	0.00068903	0.01284	0.0391068	0.374332	3.02946	33.9304
6	0.000149012	0.000153065	0.000828981	0.00574088	0.0461171	0.382689	3.03362	32.5429
7	0.000151873	0.000134945	0.00137496	0.00535083	0.0370681	0.371918	2.99946	40.0558
8	0.000118017	0.000154018	0.00120711	0.011724	0.045831	0.437991	2.99566	45.1715
9	0.000131845	0.000143051	0.00142002	0.00503111	0.036643	0.321463	3.10987	46.1754
10	0.000158072	0.000144958	0.00599003	0.00913286	0.0377901	0.321716	3.02565	36.3872
11	0.000133991	0.000143051	0.003196	0.00687194	0.0436909	0.359107	2.98617	34.5703
12	0.000129938	0.000145912	0.00177908	0.00734305	0.0433359	0.312216	3.25371	35.6952
13	0.000133991	0.000216961	0.0012908	0.00530505	0.040334	0.342626	3.28712	36.2399
14	0.000146866	0.000362873	0.00136399	0.00755811	0.049902	0.339883	3.37223	35.5598
15	0.000298977	0.000432014	0.0012331	0.00476003	0.051528	0.329303	3.14714	35.5019
16	0.000179052	0.000473022	0.00130796	0.00709414	0.048135	0.312835	3.22298	37.9751

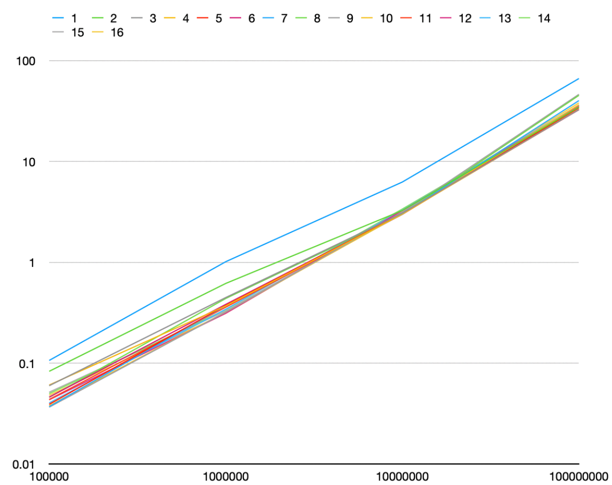
(a) 耗时

	10	100	1000	10000	100000	1000000	10000000	100000000
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	0.21	1.62	2.04	2.09	1.28	1.64	1.91	1.92
3	0.09	1.55	1.36	1.51	1.78	2.27	2.04	2.03
4	0.53	1.00	1.73	1.96	1.75	2.74	2.08	2.02
5	0.79	1.14	1.54	0.77	2.72	2.71	2.06	1.95
6	0.67	0.89	1.28	1.72	2.31	2.65	2.06	2.04
7	0.66	1.01	0.77	1.85	2.87	2.73	2.08	1.66
8	0.85	0.88	0.88	0.84	2.32	2.31	2.08	1.47
9	0.76	0.95	0.75	1.97	2.90	3.15	2.01	1.44
10	0.63	0.94	0.18	1.08	2.82	3.15	2.06	1.82
11	0.75	0.95	0.33	1.44	2.44	2.82	2.09	1.92
12	0.77	0.93	0.60	1.35	2.46	3.25	1.92	1.86
13	0.75	0.63	0.82	1.87	2.64	2.96	1.90	1.83
14	0.68	0.38	0.78	1.31	2.13	2.98	1.85	1.86
15	0.33	0.32	0.86	2.08	2.07	3.08	1.98	1.87
16	0.56	0.29	0.81	1.40	2.21	3.24	1.94	1.75

(b) 加速比



(c) 耗时数据图



(d) 耗时数据图（大数据量部分）

图 5.1: 算法一

从图表来看，算法一在 4 线程时表现最好，大约有 2 的加速比。

对于算法二，我们可以从图表数据 5 中清晰地看出它的线性收敛性[1]。并且在 2 线程时效果最好，获得 1.7 左右的加速比。

此次试验通过计算 π 运用了 OpenMP 并行框架，掌握了 OpenMP/C++ 的并行代码编写能力，体会了其中的并行思想，收获颇丰。

参考文献

[1] https://www.craig-wood.com/nick/articles/pi-chudnovsky/. Online; accessed 29 May 2020.

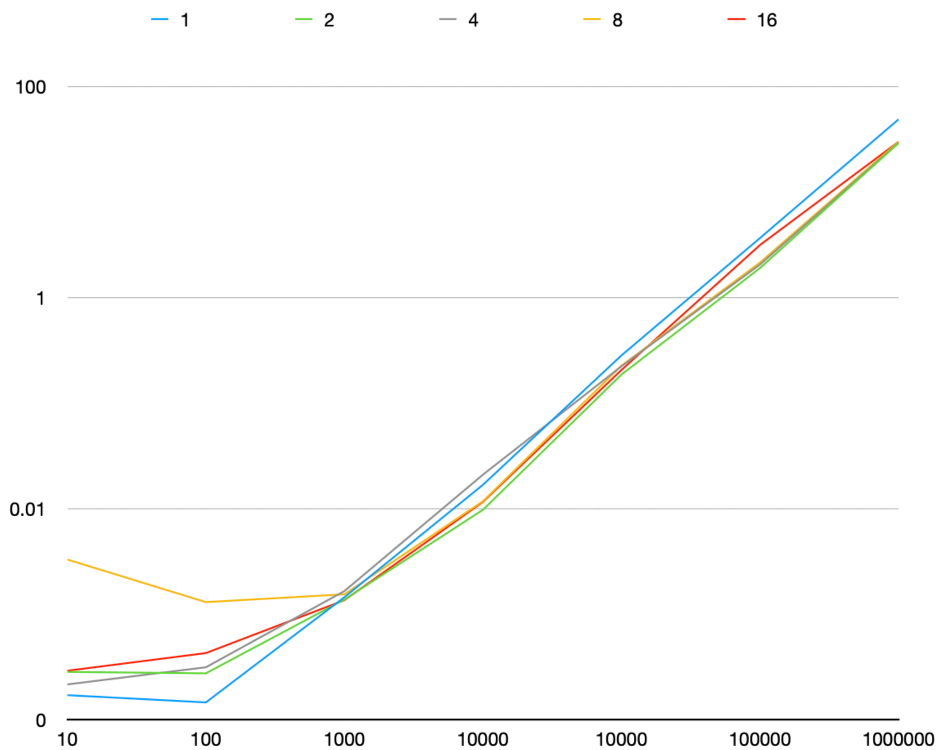
[1] 图中纵轴对数化以保持与测试时计算位数 10 倍增长的一致性。

	10	100	1000	10000	10000	100000	1000000
1	0.000170946	0.000145912	0.00144792	0.0167911	0.282426	3.65212	48.5977
2	0.000283957	0.000275135	0.00137591	0.00973797	0.186002	1.89685	28.9528
4	0.000216007	0.000313997	0.0016551	0.0209551	0.222871	2.04866	29.0932
8	0.00329089	0.0013001	0.00154185	0.0117369	0.225826	2.14144	29.7749
16	0.000291109	0.000427961	0.00135684	0.01157	0.205379	3.13773	29.839

(a) 耗时

	10	100	1000	10000	10000	100000	1000000
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	0.60	0.53	1.05	1.72	1.52	1.93	1.68
4	0.79	0.46	0.87	0.80	1.27	1.78	1.67
8	0.05	0.11	0.94	1.43	1.25	1.71	1.63
16	0.59	0.34	1.07	1.45	1.38	1.16	1.63

(b) 加速比



(c) 耗时数据图

图 5.2: 算法二

[2] Bruno Haible and Thomas Papanikolaou. Fast multiprecision evaluation of series of rational numbers. In *International Algorithmic Number Theory Symposium*, pages 338–350. Springer, 1998.