清华大学计算机系
并行程序设计

N-Body Problem

姓名	吴育昕
学号	2011011271
邮箱	ppwwyyxxc@gmail.com
时间	2012 年 8 月

目录

1 Introduction

1. 编译: 程序共有四个版本,分别为串行,MPI,OpenMP,pthread.需通过改变环境变量DEFINES分别编译.源码中提供了一个脚本make_all_version可以一次性编译出四个可执行文件. 编译并行版本的环境变量分别为DEFINES=-DUSE_OMP, -DUSE_MPI, -DUSE_PTH,单独编译时,可执行文件为main

```
$ ./make_all_version
make seq verson ...
done
make omp version ...
done
make pthread version ...
done
make mpi version ...
done
$ DEFINES=-DUSE_OMP make
[dep] main.cc ...
[dep] Gui.cc ...
[dep] Body.cc ...
[dep] utils.cc ...
[dep] common.cc ...
[dep] NBody.cc ...
[dep] NBody_Parallel.cc ...
[cc] NBody_Parallel.cc ...
[cc] NBody.cc ...
[cc] utils.cc ...
[cc] common.cc ...
[cc] Body.cc ...
[cc] Gui.cc ...
[cc] main.cc ...
Linking ...
$
```

2. 命令行参数:

```
$ ./main -h
Usage:
    ./main -b NUM [-r <switch>] [-w <switch>] [-n <NUM_OF_PROC>] [-s <SIZE>] [-t <STEP>] [-h]
Options:
    --ball=NUM,  -b      number of balls. Default: 20.
    --disp=0/1,  -d      a switch on display mode containing a big ball. Default: 1
    --wall=0/1,  -w      a switch on whether to use window border as walls. Default: 1
    --nproc=NUM, -n      number of threads(pthread only). number of CPUs by default
    --size=SIZE, -s      size of window(by pixels). Format: [width]x[height]
                         eg. 1200x800 (default)
    --step=NUM,  -t      number of loops to operate in simulation.
                         NOTE: GUI will be off to calculate time.
    --help,              -h      print help.
```

程序支持如下的命令行参数:

`--ball=NUM`指定小球个数.

`--disp=0/1,--wall=0/1`演示开关与墙壁开关.

`--nproc=NUM`指定 pthread 使用的线程数量,对其他多线程模式无效.

`--size=SIZE`指定窗口大小.

`--step=NUM`指定循环次数.

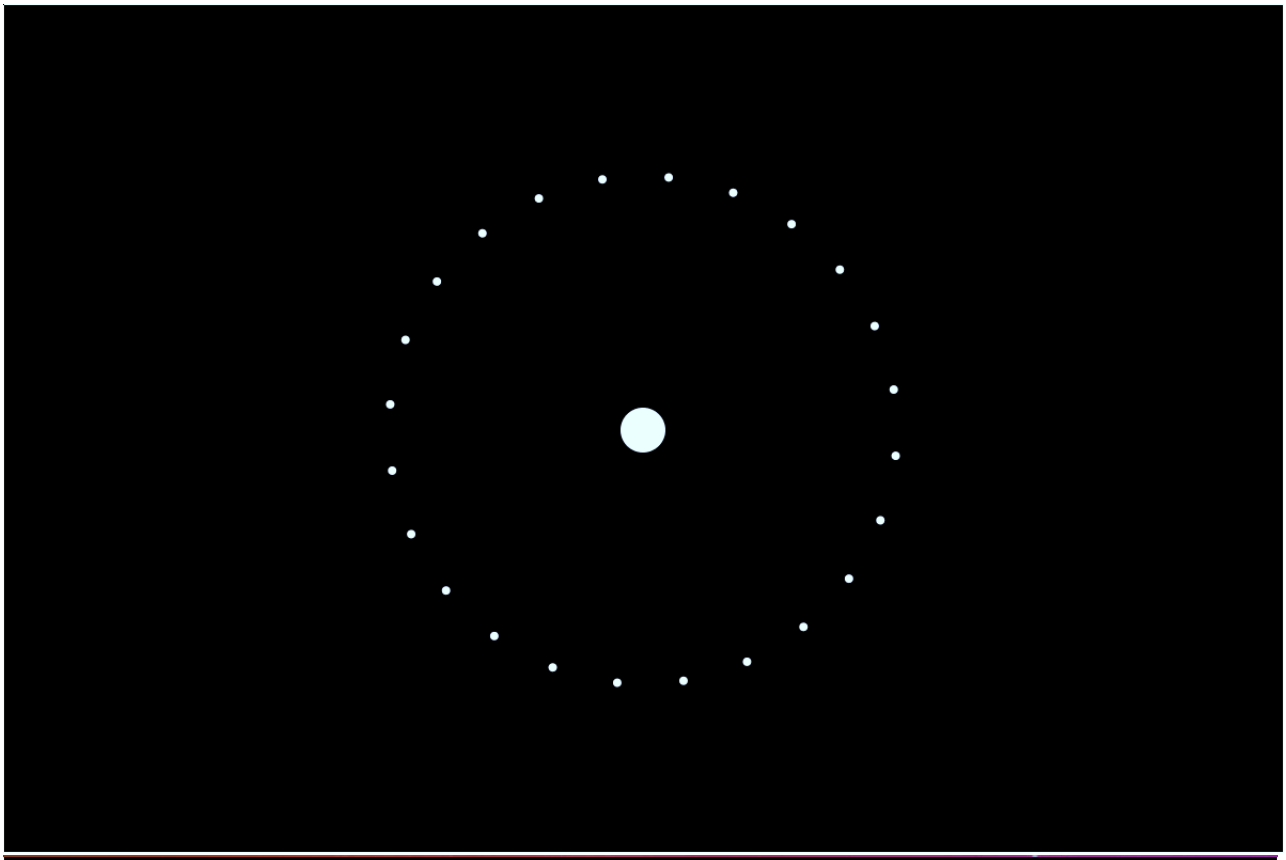
`--help`输出帮助信息.

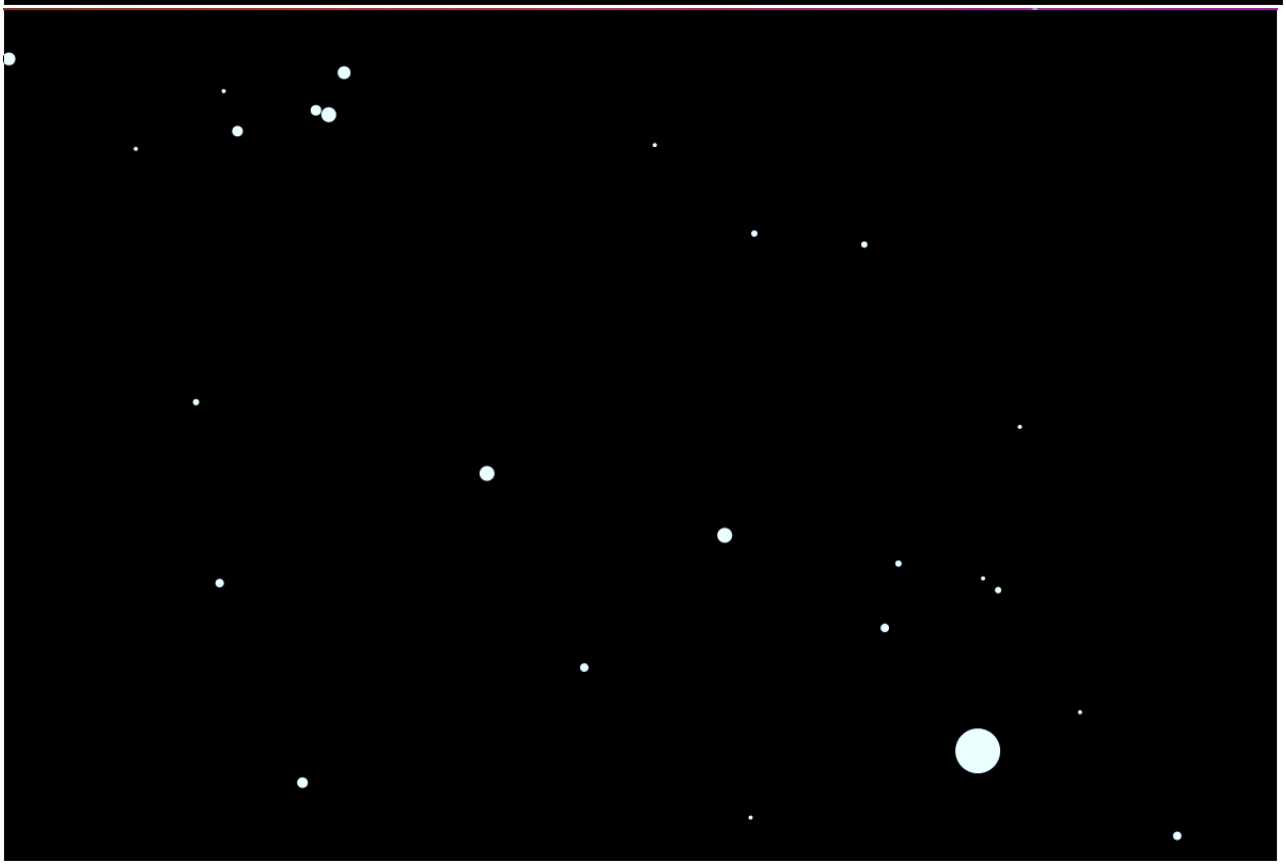
3. 测试运行:

```
$ ./main -b 200 -t 200
0.748785 seconds in total...
$
```

4. 界面展示:

```
$ ./main
10.636224 seconds in total...
$ ./main -d 0
14.245977 seconds in total...
```





在窗口中,可使用空格键进行暂停,鼠标左中右键点击屏幕,可查看距点击位置最近的小球信息.按 ESC 键退出.

2 Algorithms

N 体问题[1]是在给定一些物体的质量,位置及速度的情形下,计算未来时间中物体如何运动. 本程序是对 N 体问题的数值模拟,采用纯粹的引力方法[2]进行计算.

程序在每次循环中,首先根据牛顿引力公式 $\overrightarrow{F}_{12} = -\frac{GM_1M_2}{|\overrightarrow{r}_{12}|^2}\frac{\overrightarrow{r}_{12}}{|\overrightarrow{r}_{12}|}$ 分别算出每个物体在这一瞬间的加速度 $\overrightarrow{a}_t = \sum_{i\neq t}\frac{GM_i\overrightarrow{r}_{ti}}{|\overrightarrow{r}_{ti}|^3}$, 然后利用 $\Delta\overrightarrow{v}_t = \overrightarrow{a}_t$ 更新各物体的速度.

res/src/Body.cc

```
16 inline void cal_vel_two(Body& b1, const Body& b2, const real_t dt){
17     real_t r = distsqr(b2, b1);
18     Vec F_tmp = dt * GRAVITY / (r) * (b2.pos - b1.pos)/sqrt(r);
19     b1.v += F_tmp * b2.m;
20     return;
21 }
```

随后,程序判断是否有碰撞发生.判断依据为一个循环的 dt 时间内是否会有两球的距离小于半径之和,即 $|\overrightarrow{r}_{ti} + x\overrightarrow{(v_i - v_t)}| \leq R_i + R_t$ 是否有 $0 \leq x \leq dt$ 的解.由此找出最先与球 t 相撞的球,将两球都移动到碰撞点,计算并更新其碰撞后的速度.

res/src/Body.cc

```
23 real_t col_time(const Body &b1, const Body &b2, const real_t dt){
24     // calculate whether they will collide in dt, if yes, return collision time, else return -1
25     // equal to solve |b1.pos - b2.pos + t (b1.v - b2.v)| == b1.r + b2.r
26     // (b1.pos.x - b2.pos.x + t (b1.v.x - b2.v.x))^2 +
27     //         (b1.pos.y - b2.pos.y + t (b1.v.y - b2.v.y))^2 == (b1.r + b2.r)^2
28     real_t dpx = b1.pos.x - b2.pos.x, dpy = b1.pos.y - b2.pos.y,
29            dvx = b1.v.x - b2.v.x, dvy = b1.v.y - b2.v.y,
30        A = dvx * dvx + dvy * dvy,
31        B = 2 * dpx * dvx + 2 * dpy * dvy,
32        C = dpx * dpx + dpy * dpy - (b1.r + b2.r) * (b1.r + b2.r),
33        D = sqrt(B * B - 4 * A * C);
34     real_t t1 = ( -B + D) / 2 / A, t2 = ( -B - D) / 2 / A;
35     real_t t;
36     if ((t1 < -EPS) || (t1 > dt) || (t1 > t2)) t = t2;
37     else t = t1;
38     if ((t > -EPS) && (t < dt))
39         return t;
40     else return -1;
41 }
```

碰撞时,先将速度分解到碰撞方向与碰撞点的切线方向. 由于碰撞方向上的动量守恒,碰撞点切线方向上的速度不变,再加上能量守恒,可以类似[3]中的一维情形列出方程求解.

res/src/Body.cc

```
43 inline void col_vel_two(Body &b1, const Body &b2){
44     const Vec col = (b2.pos - b1.pos)/dist(b2, b1), mass = col / (b1.m + b2.m);
45     real_t vc1 = b1.v.dot(col), vc2 = b2.v.dot(col);
46     b1.v += 2 * b2.m * (vc2 - vc1) * mass;
47     b1.v *= COLLISION_FACTOR;
48 }
```

3 Design

程序通过编译选项-DUSE_MPI, -DUSE_OMP, -DUSE_PTH指定使用的多线程库,如果在命令行中选择了-t选项, 则程序直接进入循环计算,完成指定次数的计算后输出时间. 否则程序会初始化 gtk,利用timeout_signal方式定时调用计算函数,并在屏幕上输出图像,直至 GUI 被关闭.

当定义了USE_OMP宏时,程序调用NBody::omp()函数进行计算,函数中循环体前有#pragma omp parallel for schedule(dynamic)一行, 对下方的 for 循环自动进行了动态多线程任务调度.循环体中使用的NBody::vel_change(), NBody::collision_change()函数分别用于计算引力与碰撞对物体状态造成的改变,被所有多进程模块调用.

当定义了USE_PTH宏时,程序调用NBody::pthread()函数. 函数创建nproc个线程,每个线程每次领取pth_unit个物体的计算任务,并使用变量int now用于记录当前未领的任务. 对每个线程,在需要领取任务时将now_mutex锁定,将now更新,再解锁. 引力计算结束后,调用pthread_barrier_wait实现同步,再用同样的策略计算碰撞.

当定义了USE_MPI宏时,root 进程调用NBody::mpi_master()函数,其他进程从main()中直接调用NBody::mpi_salve() 函数. root 进程只负责进行任务调度,将任务以mpi_unit个为一组发送. 各进程接收 root 发送的 BEGIN 信息以及任务的起始编号,便开始计算,计算完成后向 root 进程发送 FINISH 信息及计算数据. 当引力计算完毕时,root 进程发送 EXIT 信息,随后各进程调用NBody::share_data() 共享 root 进程的全部数据,然后用同样方法进行碰撞计算. 随后,如果需要计算墙壁碰撞,则由 root 进程对所有物体遍历.之后由 root 进程更新位置并与其他进程广播.

4 Results and Analysis

1. 处理器核数不同时运行时间比较:

```
$ ./pthread -n 2 -b 10000 -t 10
36.127635 seconds in total...
$ ./pthread -n 4 -b 10000 -t 10
18.336705 seconds in total...
$ ./pthread -n 6 -b 10000 -t 10
12.532974 seconds in total...
$ ./pthread -n 8 -b 10000 -t 10
9.650894 seconds in total...
$ ./pthread -n 10 -b 10000 -t 10
7.912135 seconds in total...
$ ./pthread -n 12 -b 10000 -t 10
```

```
6.736707 seconds in total...

$ mpirun -n 2 ./mpi -b 10000 -t 5
36.700927 seconds in total...
$ mpirun -n 4 ./mpi -b 10000 -t 5
14.856551 seconds in total...
$ mpirun -n 8 ./mpi -b 10000 -t 5
17.607959 seconds in total...
$ mpirun -n 12 ./mpi -b 10000 -t 5
13.325384 seconds in total...
```

由以上数据可以得出,pthread 与 MPI 在核数与 CPU 数相等时效率较高,这与以往的经验吻合.

2. 各并行方式下程序运行时间比较:

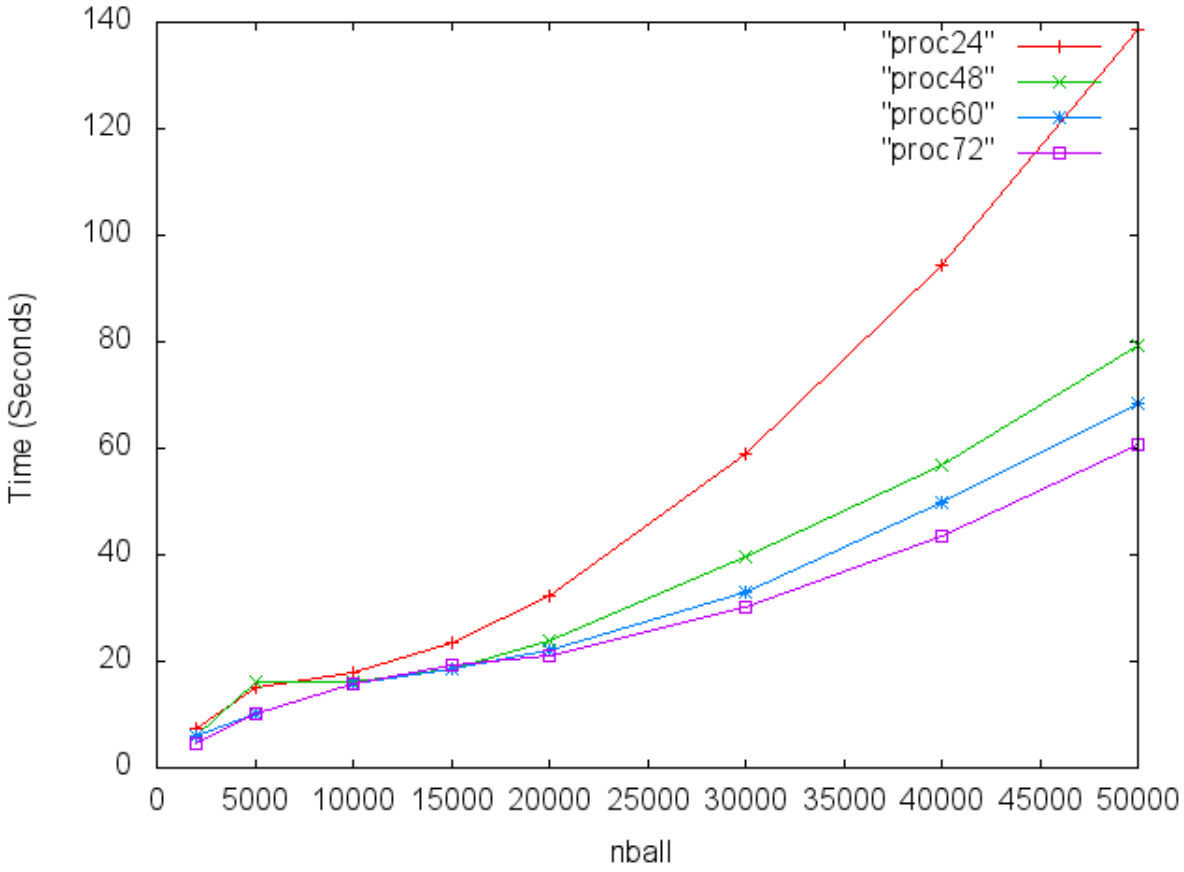
```
$ ./pthread -b 10000 -t 10
6.735550 seconds in total...
$ ./omp -b 10000 -t 10
20.149873 seconds in total...
$ mpirun -n 12 ./mpi -b 10000 -t 10
26.785754 seconds in total...
$ ./seq -b 10000 -t 10
70.296430 seconds in total...
```

本程序的并行效率比较结果表明,pthread 效率最高,其次是 OpenMP 与 MPI.这与上一个作业的结果吻合.

3. 球个数不同时运行时间比较:

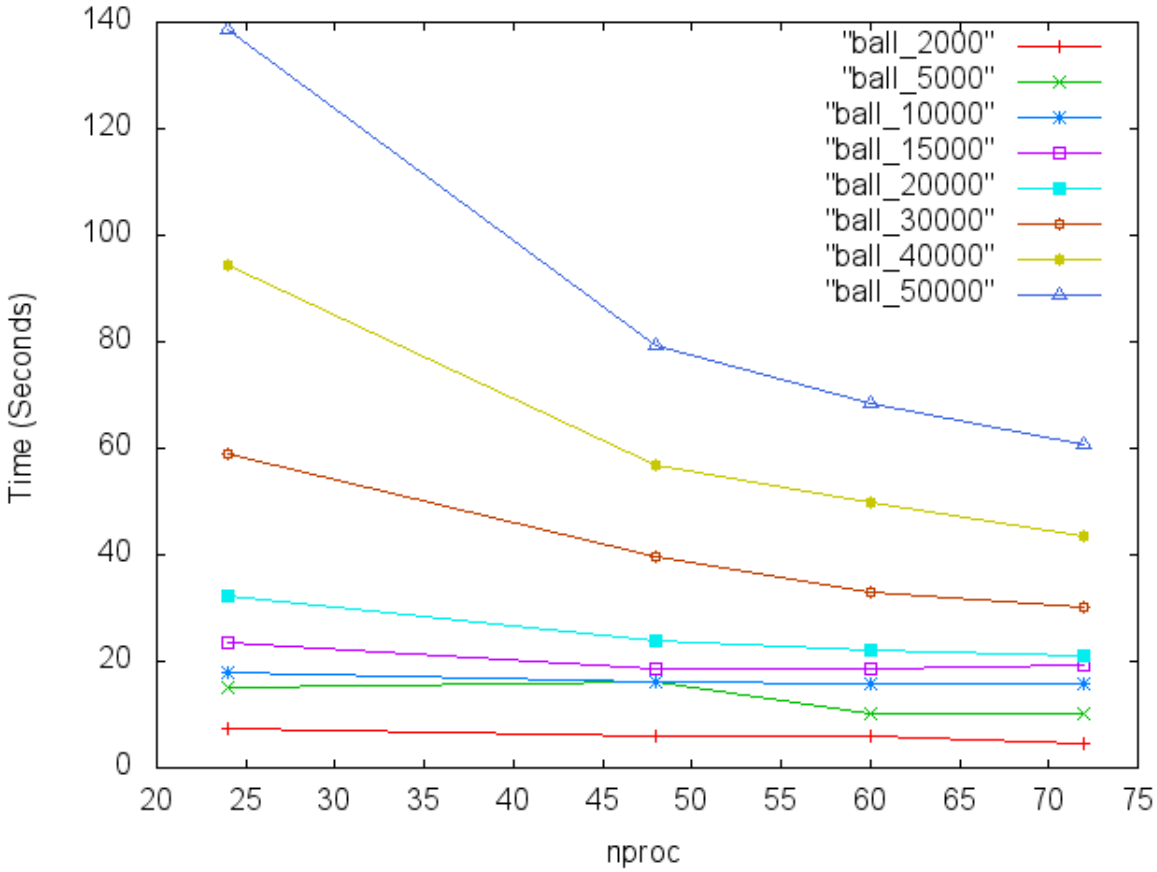
	2000	5000	10000	15000	20000	30000	40000	50000
24	7.474266	15.150455	17.909214	23.538911	32.357338	58.790446	94.269225	138.547335
48	6.104528	16.262802	16.294989	18.443990	23.920497	39.704812	56.713508	79.303986
60	5.965712	10.075106	15.898907	18.610584	22.252957	32.892880	49.755771	68.303462
72	4.468587	10.164223	15.730339	19.156377	20.973644	30.322637	43.577078	60.674458

表 1: 不同球数,处理器个数下运行时间统计(秒)



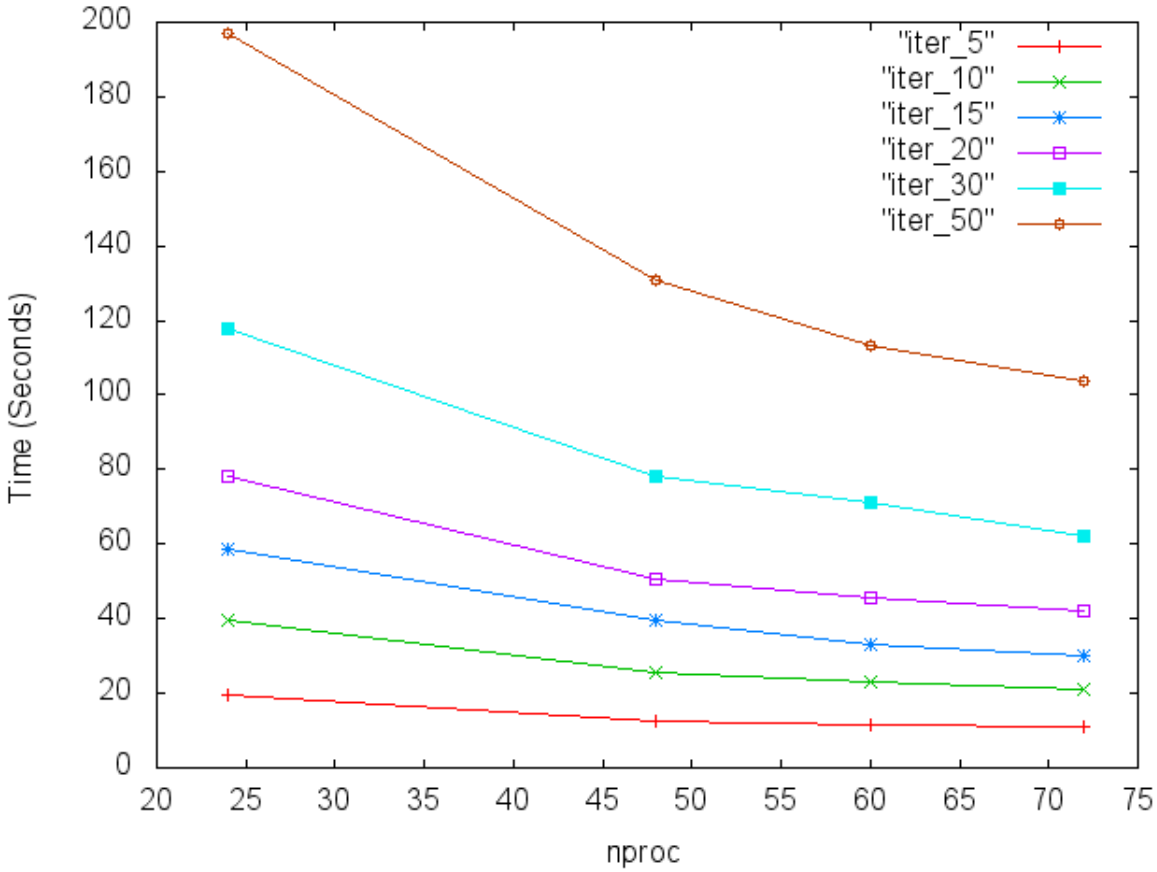
可以看出,当核数较少时,时间关于球数大约呈二次函数关系.这与算法 $O(n^2)$ 的复杂度相吻合. 核数多时,由于数据量不够大,因此对于球少的情形,并没有显著的加速.

4. 多核心大数据运行时间比较:



可以看出,数据量较小时,多线程的加速效果几乎不可见,这是由于在每个循环周期内,程序会调用两次`NBody::share_data()`来共享计算出的数据, 每次调用需要广播`MAXN * 4`个`double`,对通信负载非常大. 但由于题目的特殊性,每次计算都要求获取其余所有物体上一循环结束时的信息,因此这种通信是必不可少的, 当数据量增大时,多线程的加速效果开始变得明显.

5. 循环次数不同时运行时间比较:



从上图中可以看出,循环次数与时间大约成正比.同时还可以发现,循环次数较多时,加速比变得不明显. 这同样是因为,每次循环结束时,MPI 各进程间需要广播数据.

6. 程序效能分析:

使用 gprof,对程序(以 pthread 版本为例)效能分析部分结果如下:

```
$ gprof ./pthread gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
time   seconds   seconds  calls     name
40.01      0.34     0.34  4871892    Body::cal_vel(Body const&, double)
31.78      0.61     0.27  4088280    col_time(Body const&, Body const&, double)
9.41       0.69     0.08  5809513    overlap(Body const&, Body const&)
7.06       0.75     0.06  5657909    operator*(double, Vec const&)
5.88       0.80     0.05    15036    NBody::collision_change(int)
3.53       0.83     0.03  4324526    tooclose(Body const&, Body const&)
```

由以上结果,可以看出程序的大部分计算花费在计算引力(`cal_vel()`)与碰撞(`col_time()`,`collision_change()`)上, 这两个部分包含大量浮点运算. 上表中的`overlap()`与`tooclose()`函数是用于判断物体之间的一些相对位置关系的, 使用

它们使得我的程序展示效果更好,可以看出明显的引力与碰撞效果,没有太多的异常运动,但其中包含的浮点运算(见2节)消耗了不少资源.

5 Summary & Experience

在第二次作业中,我熟悉了 gtkmm 的开发,但由于清华 FIT 楼集群计算机没有 gtkmm 相关库,在第三次作业中我便使用了 Xlib. Xlib 功能太弱,gtk 所支持的信号功能对于这次的程序很重要,因此此次我使用 gtk.

在并行方面,由于 N 体问题每个计算都涉及很多其他数据的参与, 可以独立并行计算的部分不多,因此对并行算法的设计中不得不在效率与准确度之间做一些取舍. 因此大家的程序效率,演示效果都有不少差别.如我的 MPI 程序,就因计算时的几次数据同步耗去了大量时间.

上次的作业中,算法已经给出,只需设计并行部分.而这次的作业中算法是个难点,较好的选择常数,处理碰撞, 使得 demo 中不出现不科学的运动模式花了我大多数的时间.

这次作业模拟的是物理问题,引力与碰撞的相关问题也让我温习了物理知识,设计常量的过程也让我感受到了宇宙常数的难以选择,不合适的常数总使得物体的运动十分不规则,让我更深刻的体会到了关于神秘宇宙常数的“Fine-tuned Universe”的论断[4].

6 References

[1] *Wikipedia page for N-Body Problem*. URL: http://en.wikipedia.org/wiki/N-body_problem.

[2] *Wikipedia page for N-Body Simulation*. URL: http://en.wikipedia.org/wiki/N-body_simulation.

[3] *Wikipedia page for Elastic Collision*. URL: http://en.wikipedia.org/wiki/Elastic_collision.

[4] *Wikipedia page for Fine-tuned Universe*. URL: http://en.wikipedia.org/wiki/Fine-tuned_Universe.