**高性能并行计算第4次作业**

姓名：姚虎成 学号：2020317110033

**代码地址：**

1、六种并行计算方法计算Pi的C语言程序路径

/home2/2020317110033/openmp\_4/pi\_parallel\_all.c

/home2/2020317110033/openmp\_4/pi\_sh

2、三种并行计算方法计数组的C语言程序路径

/home2/2020317110033/openmp\_4/k\_parallel\_all.c

/home2/2020317110033/openmp\_4/k\_sh

**实验结果：**

**1、**在计算pi的C程序时，由于计算时间过短，因此在代码实现过程中将steps扩大至1亿次。表1.1-1.6是采用并行域、Padding技术、同步制导语句包括Critical技术、Atomic技术、Barrier技术以及For Reduction规约技术等7种方法并行计算Pi。

表1.1分别展示了随着CPU数目的变化，时间，强扩展效率，以及加速比的变化，可以看出，随着CPU数目的增加，并行效率以及加速比甚至减少，其结果也不稳定。可能的原因是使用的服务器使用的Docker容器技术，在实际使用CPU数目过程中，可能CPU实际数目并没有那么多，导致并行效率的降低。还有可能是同一时间在服务器运行脚本还有别的同学，导致资源争抢。

**表1.1 并行域计算Pi**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 1.680021 | 100.00% | 1.000000 |
| 2 | 1.872977 | 44.85% | 0.896979 |
| 3 | 2.778099 | 20.16% | 0.604738 |
| 4 | 3.172086 | 13.24% | 0.529627 |
| 5 | 1.358635 | 24.73% | 1.236551 |
| 6 | 3.576840 | 7.83% | 0.469694 |
| 7 | 2.130868 | 11.26% | 0.788421 |
| 8 | 1.276316 | 16.45% | 1.316305 |
| 9 | 2.042534 | 9.14% | 0.822518 |
| 10 | 2.541952 | 6.61% | 0.660918 |

表1.2 是采用了Padding技术，利用了一个二维数组来存储积分加和结果，可以看出，随着CPU数目的增加，并行效率基本都维持在99%，CPU数量增多至7个后，并行效率轻微下降，可能是因为通信开销逐渐变大。加速比的提升接近于线性加速比，加速结果表现良好。

**表1.2 Padding技术计算Pi**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 1.679829 | 100.00% | 1.000000 |
| 2 | 0.840302 | 99.95% | 1.999078 |
| 3 | 0.560642 | 99.88% | 2.996260 |
| 4 | 0.420315 | 99.91% | 3.996597 |
| 5 | 0.335990 | 99.99% | 4.999643 |
| 6 | 0.279982 | 100.00% | 5.999780 |
| 7 | 0.240307 | 99.86% | 6.990343 |
| 8 | 0.210169 | 99.91% | 7.992745 |
| 9 | 0.187005 | 99.81% | 8.982811 |
| 10 | 0.168364 | 99.77% | 9.977386 |

表1.3是采用了同步制导语句中的critical技术，critical制导语句中的代码一次只能执行一个线程，可以解决积分加和问题。

可以看出，随着CPU数目的增加，其加速比也接近于线性加速比，并行效率基本维持在100%附近，加速效果表现优秀。甚至有几次超过了100%，这可能是计算机缓存技术导致其产生了超线性加速，随着CPU数目的进一步增多，其并行效率有轻微下降趋势，这可能是随着CPU数目增多，通信的开销逐渐增多。

**表1.3 Critical技术计算Pi**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 1.679451 | 100.00% | 1.000000 |
| 2 | 0.839931 | 99.98% | 1.999511 |
| 3 | 0.559814 | 100.00% | 3.000017 |
| 4 | 0.420288 | 99.90% | 3.995953 |
| 5 | 0.335705 | 100.06% | 5.002755 |
| 6 | 0.279893 | 100.01% | 6.000324 |
| 7 | 0.240193 | 99.89% | 6.992083 |
| 8 | 0.210351 | 99.80% | 7.984023 |
| 9 | 0.187047 | 99.76% | 8.978746 |
| 10 | 0.168822 | 99.48% | 9.948067 |

表1.4是跟第三种方法类似，采用了同步制导语句Atomic语句技术实现并行计算，atomic指导语句指定特定的存储单元将被原子更新，即atomic中的操作会被翻译为原子操作，同一时间只能有一个线程执行他，其他的线程需要等待。用于积分加和。

可以看出，随着CPU数目的增多，其并行效率和加速比变化跟critical类似，并行效率基本稳定100%，加速比接近线性上升。加速结果表现优秀。

**表1.4** **Atomic 技术计算Pi**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 1.680018 | 100.00% | 1.000000 |
| 2 | 0.840495 | 99.94% | 1.998845 |
| 3 | 0.560219 | 99.96% | 2.998859 |
| 4 | 0.420162 | 99.96% | 3.998503 |
| 5 | 0.335855 | 100.04% | 5.002213 |
| 6 | 0.279945 | 100.02% | 6.001237 |
| 7 | 0.240089 | 99.96% | 6.997479 |
| 8 | 0.210223 | 99.89% | 7.991597 |
| 9 | 0.187102 | 99.77% | 8.979170 |
| 10 | 0.168198 | 99.88% | 9.988365 |

表1.5跟前面的Atomic和Critical语句类似，采用了同步制导语句Barrier技术设置障碍进行等待来实现并行计算。

可以看出，使得CPU数目的增多，其并行效率基本稳定在100%，加速比也接近理想的线性加速比，并行速度有明显提升。

**表1.5 Barrier技术计算Pi**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 1.674773 | 100.00% | 1.000000 |
| 2 | 0.839937 | 99.70% | 1.993927 |
| 3 | 0.559130 | 99.84% | 2.995321 |
| 4 | 0.419086 | 99.91% | 3.996251 |
| 5 | 0.336375 | 99.58% | 4.978893 |
| 6 | 0.279812 | 99.76% | 5.985348 |
| 7 | 0.240012 | 99.68% | 6.977876 |
| 8 | 0.210054 | 99.66% | 7.973064 |
| 9 | 0.186728 | 99.66% | 8.969060 |
| 10 | 0.168184 | 99.58% | 9.957976 |

表1.6采用了firstprivate方法，通过firistprivate（sum）将sum定义成每个线程的私有变量，使得循环并行执行的时候，将sum分给各个线程私有化。

可以看出，使得CPU数目的增多，其并行效率基本稳定在100%，加速比也接近理想的线性加速比，并行速度有明显提升。

**表1.6 Firstprivate技术计算Pi**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 1.679681 | 100.00% | 1.000000 |
| 2 | 0.840118 | 99.97% | 1.999339 |
| 3 | 0.560381 | 99.91% | 2.997393 |
| 4 | 0.420269 | 99.92% | 3.996679 |
| 5 | 0.335905 | 100.01% | 5.000472 |
| 6 | 0.279870 | 100.03% | 6.001656 |
| 7 | 0.240071 | 99.95% | 6.996607 |
| 8 | 0.210589 | 99.70% | 7.976125 |
| 9 | 0.186926 | 99.84% | 8.985829 |
| 10 | 0.168257 | 99.83% | 9.982839 |

表1.6，使用openMP中的for循环配合reduction规约操作实现并行计算加速。相比之前的atomic和critical，加入了reduction（+：sum）将各个线程分开，省去了critical和atomic语句，其并行效率基本稳定在100%，加速比也接近理想的线性加速比。并行速度明显提升。

**表1.7 Reduction技术计算Pi**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 1.680068 | 100.00% | 1.000000 |
| 2 | 0.840574 | 99.94% | 1.998714 |
| 3 | 0.559865 | 100.03% | 3.000845 |
| 4 | 0.420159 | 99.97% | 3.998648 |
| 5 | 0.336072 | 99.98% | 4.999124 |
| 6 | 0.279975 | 100.01% | 6.000773 |
| 7 | 0.240228 | 99.91% | 6.993637 |
| 8 | 0.210293 | 99.86% | 7.989187 |
| 9 | 0.186931 | 99.86% | 8.987616 |
| 10 | 0.168347 | 99.80% | 9.979780 |

**2、**表2.1-2.3是利用Padding、Critical、Reduction三种技术计算数组**K=**的结果。

**表2.1 Padding技术计算数组K项之和**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 0.490997 | 100.00% | 1.000000 |
| 2 | 0.245924 | 99.83% | 1.996536 |
| 3 | 0.164001 | 99.80% | 2.993855 |
| 4 | 0.127461 | 96.30% | 3.852133 |
| 5 | 0.098406 | 99.79% | 4.989521 |
| 6 | 0.084463 | 96.89% | 5.813165 |
| 7 | 0.070790 | 99.08% | 6.935914 |
| 8 | 0.063950 | 95.97% | 7.677841 |
| 9 | 0.055263 | 98.72% | 8.884774 |
| 10 | 0.050271 | 97.67% | 9.767085 |

表2.1是利用Padding技术，计算了数组K的加和，随着CPU数量的增加，并行效率表现良好，加速比基本上呈线性加速，有轻微不稳定，可能是因为同一时间有同学提交任务。

表2.2 是利用Critical技术，计算了数组K的加和，随着CPU数量的增加，结果基本和padding结果类似。并行效率表现良好，加速比基本上呈线性加速，有轻微不稳定，中间有一次在8个CPU时，加速效果不明显，可能是因为同一时间有同学提交任务。

表2.3 是利用Reduction技术，计算了数组K的加和，并行结果类似利用Padding和Critical技术的结果。随着CPU数量的增加，并行效率表现良好，加速比基本上呈线性加速，结果也有轻微不稳定。

**表2.2 Critical技术计算数组K项之和**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 0.400868 | 100.00% | 1.000000 |
| 2 | 0.203758 | 98.37% | 1.967370 |
| 3 | 0.136826 | 97.66% | 2.929764 |
| 4 | 0.101264 | 98.97% | 3.958647 |
| 5 | 0.083380 | 96.15% | 4.807714 |
| 6 | 0.068710 | 97.24% | 5.834197 |
| 7 | 0.058018 | 98.70% | 6.909334 |
| 8 | 0.055077 | 90.98% | 7.278328 |
| 9 | 0.044574 | 99.93% | 8.993382 |
| 10 | 0.040179 | 99.77% | 9.977121 |

**表2.3 Reduction技术计算数组K项之和**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 0.409546 | 100.00% | 1.000000 |
| 2 | 0.208637 | 98.15% | 1.962964 |
| 3 | 0.137122 | 99.56% | 2.986734 |
| 4 | 0.102388 | 100.00% | 3.999929 |
| 5 | 0.085485 | 95.82% | 4.790836 |
| 6 | 0.068486 | 99.67% | 5.980034 |
| 7 | 0.062465 | 93.66% | 6.556452 |
| 8 | 0.062532 | 81.87% | 6.549402 |
| 9 | 0.045561 | 99.88% | 8.988890 |
| 10 | 0.041005 | 99.88% | 9.987755 |

**实验分析：**

在计算Pi的程序中，由于其计算时间过段，将积分中的steps提高至1亿。使用了并行域、Padding、Critical、Atomic、Firstprivate、Reduction这七种方法对Pi进行并行计算，并对其运行时间，并行效率和加速比做了统计计算。具体脚本见附录。

在实现一次打印从1核到10核不同计算方法的运行时间，我们使用了shell脚本，通过在脚本种设置一个列表，利用shell脚本编程中for循环，进行设置环境变量OMP\_NUM\_THREADS，具体脚本见附录。

对于以上七种方法进行对比，我们发现利用相比串行计算，除了使用并行域计算会出现并行效率和加速比减少情况，其他方法如使用atomic、critical、barrier、reduction、firstprivate其并行速度都有明显提升，加速比显著提高，并行效率基本都维持99%。像critical、atomic、firstprivate、redduction方法进行并行计算时，会偶尔出现并行效率突然减少，以及加速比增加缓慢的情况，原因可能有三：首先可能是由于同一时间段还有别的同学正在提交任务。还有可能是由于集群是使用Docker虚拟技术，导致实际使用时，并没有那么多CPU分配给我提交的任务使用，有一定的等待时间。最后可能是随着CPU数量的增加，系统之间通信开销变大，导致加速比不能线性增加。

**附录：**

代码1、七种方法并行计算Pi值的C语言程序代码

#include<stdio.h>

#include<omp.h>

#include <time.h>

#define PAD 10

static long num\_steps=100000000;

double step;

//0

//computing pi serially

void pi\_serial(){

int i;

double x, pi, sum = 0.0;

clock\_t start, end;

start = clock();

step = 1.0/(double) num\_steps;

for (i=1;i<= num\_steps; i++){

x = (i-0.5)\*step;

sum = sum + 4.0/(1.0+x\*x);

}

pi = step \* sum;

end = clock();

printf("the serial program of calculating pi running for %.8fs\n", (double)(end-start)/CLOCKS\_PER\_SEC);

}

//01

//computing pi parallel

void pi\_parallel(){

int NUM\_THREADS = omp\_get\_max\_threads();

int i;

double x,pi,sum[NUM\_THREADS],start,end;

step=1.0/(double)num\_steps;

start=omp\_get\_wtime();

#pragma omp parallel

{

int i;

double x;

int id;

id=omp\_get\_thread\_num();

for(i=id,sum[id]=0.0;i<num\_steps;i=i+NUM\_THREADS){

x=(i-0.5)\*step;

sum[id]+=4.0/(1.0+x\*x);

}

} end=omp\_get\_wtime();

for(i=0,pi=0.0;i<NUM\_THREADS;i++) pi += sum[i]\*step;

//printf("the true value of pi is %f\n ",pi);

printf("the parallel program running for %.8g s\n ",end-start);

}

//02

//computing pi parallel using padding

void pi\_parallel\_pad(){

int i;

int NUM\_THREADS = omp\_get\_max\_threads();

double x,pi,sum[NUM\_THREADS][PAD],start,end;

step = 1.0/(double)num\_steps;

start=omp\_get\_wtime();

omp\_set\_num\_threads(NUM\_THREADS);

#pragma omp parallel private(i)

{

double x;

int i,id; id = omp\_get\_thread\_num();

for (i=id, sum[id][0]=0.0;i< num\_steps; i=i+NUM\_THREADS){

x = (i-0.5)\*step;

sum[id][0] += 4.0/(1.0+x\*x);

}

} end=omp\_get\_wtime();

for(i=0, pi=0.0;i<NUM\_THREADS;i++){

pi += sum[i][0]\*step;

}

//printf("the padding true value of pi is %f\n ",pi);

printf("the padding parallel program running for %.8g s\n ",end-start);

}

//03

//critical

void pi\_critical(){

int NUM\_THREADS = omp\_get\_max\_threads();

double pi,start,end;

step=1.0/(double)num\_steps;

start=omp\_get\_wtime();

#pragma omp parallel

{

double x,sum;

int id,i; id=omp\_get\_thread\_num();

for(i=id,sum=0.0;i<num\_steps;i=i+NUM\_THREADS){

x=(i+0.5)\*step;

sum+=4.0/(1.0+x\*x);

}

#pragma omp critical

pi+=sum\*step;

} end=omp\_get\_wtime();

//printf("the true value of pi using critical is %f\n ",pi);

printf("the parallel program using critical running for %.8g s\n ",end-start);

}

//04

//atomic

void pi\_atomic(){

int NUM\_THREADS = omp\_get\_max\_threads();

double pi,start,end;

step=1.0/(double)num\_steps;

start=omp\_get\_wtime();

#pragma omp parallel

{

double x,sum;

int id,i; id=omp\_get\_thread\_num();

for(i=id,sum=0.0;i<num\_steps;i=i+NUM\_THREADS){

x=(i+0.5)\*step;

sum+=4.0/(1.0+x\*x);

}

#pragma omp atomic

pi+=sum\*step;

} end=omp\_get\_wtime();

//printf("the true value of pi using atomic is %f\n ",pi);

printf("the parallel program using atomic running for %.8g s\n ",end-start);

}

//05

//barrier

void pi\_barrier(){

int i;

int NUM\_THREADS = omp\_get\_max\_threads();

double x,pi,sum[NUM\_THREADS][PAD],start,end;

step = 1.0/(double)num\_steps;

start=omp\_get\_wtime();

omp\_set\_num\_threads(NUM\_THREADS);

#pragma omp parallel private(i)

{

double x;

int i,id; id = omp\_get\_thread\_num();

for (i=id, sum[id][0]=0.0;i< num\_steps; i=i+NUM\_THREADS){

x = (i-0.5)\*step;

sum[id][0] += 4.0/(1.0+x\*x);

}

} end=omp\_get\_wtime();

#pragma omp barrier

for(i=0, pi=0.0;i<NUM\_THREADS;i++){

pi += sum[i][0]\*step;

}

//printf("the padding true value of pi is %f\n ",pi);

printf("the parallel program using barrier running for %.8g s\n ",end-start);

}

//06

//firstprivate

void pi\_firstprivate(){

int NUM\_THREADS = omp\_get\_max\_threads();

double pi=0.0,start,end,sum=0.0;

step = 1.0/(double)num\_steps;

start = omp\_get\_wtime();

#pragma omp parallel firstprivate(sum)

{

double x;

int i,id; id = omp\_get\_thread\_num();

for(i=id;i<num\_steps;i=i+NUM\_THREADS){

x=(i-0.5)\*step;

sum+=4.0/(1.0+x\*x);

}

pi+=sum\*step;

}end=omp\_get\_wtime();

//printf("the true value of pi using firstprivate is %f\n ",pi);

printf("the parallel program using firstprivate running for %.8g s\n ",end-start);

}

//07

//for reduction

void pi\_reduction(){

int NUM\_THREADS = omp\_get\_max\_threads();

int i;

double pi,sum=0.0,start,end;

step = 1.0/(double)num\_steps;

start=omp\_get\_wtime();

#pragma omp parallel

{

double x;

int i;

#pragma omp for reduction(+:sum)

for (i=1;i< num\_steps;i++){

x = (i-0.5)\*step;

sum += 4.0/(1.0+x\*x);

}

pi = sum\*step;

}end=omp\_get\_wtime();

//printf("the true value of pi using retuction is %f\n ",pi);

printf("the parallel program using reduction running for %.8g s\n ",end-start);

}

//08

// section

void pi\_section(){

int NUM\_THREADS = omp\_get\_max\_threads();

double pi,start,end;

double sum1,sum2,sum3,sum4;

step = 1.0/(double)num\_steps;

start=omp\_get\_wtime();

#pragma omp parallel

{

printf("pid:%d,tid=%ld\n",omp\_get\_thread\_num());

#pragma omp sections

{

#pragma omp section

{

double x=0.0;

double i;

for (i=1;i< num\_steps/4;i++){

x = (i-0.5)\*step;

sum1 += 4.0/(1.0+x\*x);

}

printf("section1 %d",sum1);

printf("section 1,threadid=%d\n",omp\_get\_thread\_num());

}

#pragma omp section

{

double x=0.0;

double i;

for (i=num\_steps/4;i< num\_steps/2;i++){

x = (i-0.5)\*step;

sum2 += 4.0/(1.0+x\*x);

}

printf("section2 %d",sum2);

printf("section 2,threadid=%d\n",omp\_get\_thread\_num());

}

#pragma omp section

{

double x=0.0;

double i;

for (i=num\_steps/2;i< 3\*num\_steps/4;i++){

x = (i-0.5)\*step;

sum3 += 4.0/(1.0+x\*x);

}

//printf("section3 %d",sum3);

//printf("section 3,threadid=%d\n",omp\_get\_thread\_num());

}

#pragma omp section

{

double x=0.0;

double i;

for (i=num\_steps\*3/4;i< num\_steps;i++){

x = (i-0.5)\*step;

sum4 += 4.0/(1.0+x\*x);

}

//printf("section4 %d",sum4);

//printf("section 4,threadid=%d\n",omp\_get\_thread\_num());

}

}

}

pi=(sum1+sum2+sum3+sum4)\*step;

end=omp\_get\_wtime();

printf("the true value of pi using section is %f\n ",pi);

printf("the parallel program using section running for %.8g s\n ",end-start);

}

void main(){

pi\_serial();

pi\_parallel();

pi\_parallel\_pad();

pi\_critical();

pi\_atomic();

pi\_barrier();

pi\_firstprivate();

pi\_reduction();

//pi\_section();

}

2、批量计算不同线程的shell脚本

#!/bin/bash

#this script export OMP\_NUM\_THREADS.

list="1 2 3 4 5 6 7 8 9 10"

cd /home2/2020317110033/openmp\_4

for num\_threads in $list

do

echo "NUM\_THREADS:$num\_threads "

export OMP\_NUM\_THREADS=$num\_threads

./pi\_parallel\_all

Done

3、三种方法计算数组K项之和

#include<stdio.h>

#include<omp.h>

#include <time.h>

#define PAD 10

static long N=100000000;

//serial k

void k\_serial(){

long i,j,k;

double sum;

clock\_t start,end;

start=clock();

for(i = 1 ; i < N ; i++){

j=i+1;k=i+1;

sum+=(i+j+k)/2;

}

end=clock();

printf("sum:%.8f",sum);

printf("the serial program of calculating k running for %.8fs\n",(double)(end-start)/CLOCKS\_PER\_SEC);

}

//using padding

void k\_padding(){

int NUM\_THREADS = omp\_get\_max\_threads();

int i;

double temp[NUM\_THREADS][PAD],start, end;

double sum;

start = omp\_get\_wtime();

#pragma omp parallel firstprivate(i)

{

double x;

long i, j, k;

int id; id = omp\_get\_thread\_num();

for (i = id, temp[id][0]=0.0;i < N-2; i = i + NUM\_THREADS)

{

j = i+1; k = i+2 ;

temp[id][0] += (long)(i + j + k)/2;

}

}

end = omp\_get\_wtime();

for(i=0, sum=0.0;i<NUM\_THREADS;i++){

sum += temp[i][0];

}

printf("sum:%.8f",sum);

printf("the parallel program using padding running for %.8g s\n ",end-start);

}

//k critical parallel

void k\_critical(){

int NUM\_THREADS = omp\_get\_max\_threads();

double sum,start, end;

long temp;

start = omp\_get\_wtime();

#pragma omp parallel firstprivate(temp)

{

double x;

long i, j, k;

int id; id = omp\_get\_thread\_num();

for (i = id,temp=0.0; i < N-2; i = i + NUM\_THREADS)

{

j = i+1; k = i+2 ;

temp += (i + j + k)/2;

}

#pragma omp critical

sum += temp;

}

end = omp\_get\_wtime();

printf("sum:%.8f",sum);

printf("the parallel program using critical running for %.8g s\n ",end-start);

}

void k\_reduction(){

int NUM\_THREADS = omp\_get\_max\_threads();

double sum,start, end;

start = omp\_get\_wtime();

#pragma omp parallel reduction(+:sum)

{

double x;

long i, j, k;

int id; id = omp\_get\_thread\_num();

for (i = id; i < N-2; i = i + NUM\_THREADS)

{

j = i+1; k = i+2 ;

sum += (i + j + k)/2;

}

}

end = omp\_get\_wtime();

printf("sum:%.8f",sum);

printf("the parallel program using reduction running for %.8g s\n ",end-start);

}

void main(){

k\_serial();

k\_padding();

k\_critical();

k\_reduction();

}

4、批量计算不同线程下的K项数组之和

#!/bin/bash

#this script export OMP\_NUM\_THREADS.

list="1 2 3 4 5 6 7 8 9 10"

cd /home2/2020317110033/openmp\_4

for num\_threads in $list

do

echo "NUM\_THREADS:$num\_threads "

export OMP\_NUM\_THREADS=$num\_threads

./k\_parallel\_all

done