**高性能并行计算第4次作业**

姓名：代宏刚 学号：2020317110061

**代码地址：**

**（1）.计算pi**

**① 域并行计算pi**

1. #include <stdio.h>
2. #include <omp.h>
3. **static** **long** num\_steps = 100000000;
4. **double** step;
5. **double** start, end;
6. **int** main() {
7. start = omp\_get\_wtime();
8. **int** i;
9. **double** x, pi, sum[omp\_get\_max\_threads()];
10. step = 1.0/(**double**)num\_steps;
11. #pragma omp parallel private(x,i)
12. {
13. **int** id;
14. id = omp\_get\_thread\_num();
15. **for**(i=id,sum[id]=0.0; i<num\_steps; i+=omp\_get\_num\_threads()) {
16. x = (i-0.5) \* step;
17. sum[id] += 4.0/(1.0+x\*x);
18. }
19. }
20. **for**(i=0,pi=0.0; i<omp\_get\_max\_threads(); i++)
21. pi += sum[i]\*step;
22. end = omp\_get\_wtime();
23. printf("%d\t%fs\t",omp\_get\_max\_threads(),end-start);
24. **return** 0;
25. }

②计算pi - padding

1. #include <stdio.h>
2. #include <omp.h>
3. #define PAD 8
4. **static** **long** num\_steps = 100000000;
5. **double** step;
6. **double** start, end;
7. **int** main() {
8. start = omp\_get\_wtime();
9. **int** i;
10. **double** x, pi, sum[omp\_get\_max\_threads()][PAD];
11. step = 1.0/(**double**)num\_steps;
12. #pragma omp parallel private(x,i)
13. {
14. **int** id;
15. id = omp\_get\_thread\_num();
16. **for**(i=id,sum[id][0]=0.0; i<num\_steps; i+=omp\_get\_num\_threads()) {
17. x = (i-0.5) \* step;
18. sum[id][0] += 4.0/(1.0+x\*x);
19. }
20. }
21. **for**(i=0,pi=0.0; i<omp\_get\_max\_threads(); i++)
22. pi += sum[i][0]\*step;
23. end = omp\_get\_wtime();
24. printf("%fs\t",end-start);
25. **return** 0;
26. }

③计算pi - atomic

1. #include <stdio.h>
2. #include <omp.h>
3. **static** **long** num\_steps = 100000000;
4. **double** step;
5. **int** main() {
6. **int** NUM\_THREADS = omp\_get\_max\_threads();
7. **double** start, end;
8. start = omp\_get\_wtime();
9. **double** pi;
10. step = 1.0/(**double**)num\_steps;
11. omp\_set\_num\_threads(NUM\_THREADS);
12. #pragma omp parallel
13. {
14. **double** x, sum;
15. **int** id , i;
16. id = omp\_get\_thread\_num();
17. **for**(i=id,sum=0.0; i<num\_steps; i+=NUM\_THREADS) {
18. x = (i+0.5) \* step;
19. sum += 4.0/(1.0+x\*x);
20. }
21. #pragma omp atomic
22. pi += sum \* step;
23. }
24. end = omp\_get\_wtime();
25. printf("%f\t",end-start);
26. **return** 0;
27. }

④.计算pi – critic

1. #include <stdio.h>
2. #include <omp.h>
3. **static** **long** num\_steps = 100000000;
4. **double** step;
5. **int** main() {
6. **int** NUM\_THREADS = omp\_get\_max\_threads();
7. **double** start, end;
8. start = omp\_get\_wtime();
9. **double** pi;
10. step = 1.0/(**double**)num\_steps;
11. omp\_set\_num\_threads(NUM\_THREADS);
12. #pragma omp parallel
13. {
14. **double** x, sum;
15. **int** id , i;
16. id = omp\_get\_thread\_num();
17. **for**(i=id,sum=0.0; i<num\_steps; i+=NUM\_THREADS) {
18. x = (i+0.5) \* step;
19. sum += 4.0/(1.0+x\*x);
20. }
21. #pragma omp critical
22. pi += sum \* step;
23. }
24. end = omp\_get\_wtime();
25. printf("%f\t",end-start);
26. **return** 0;
27. }

⑤.计算pi – parallel for

1. #include <stdio.h>
2. #include <omp.h>
3. **static** **long** num\_steps = 100000000;
4. **double** step;
5. **int** main() {
6. **int** NUM\_THREADS = omp\_get\_max\_threads();
7. **double** start, end;
8. start = omp\_get\_wtime();
9. **double** pi,sum = 0.0;
10. step = 1.0/(**double**)num\_steps;
11. omp\_set\_num\_threads(NUM\_THREADS);
12. #pragma omp parallel
13. {
14. **double** x;
15. **int** i;
16. #pragma omp for reduction(+:sum)
17. **for**(i=1; i<num\_steps; i++) {
18. x = (i+0.5) \* step;
19. sum += 4.0/(1.0+x\*x);
20. }
21. pi =  sum \* step;
22. }
23. end = omp\_get\_wtime();
24. printf("%f\n",end-start);
25. **return** 0;
26. }

（2）计算数列和

①.多线程计算数列

1. #include <stdio.h>
2. #include <omp.h>
3. #include <malloc.h>
4. #define N 100000000
5. **int** arr[N];
6. **int** main(){
7. **double** start, end;
8. **int** NUM\_THREADS = omp\_get\_max\_threads();
9. **int** i = 0;
10. **long** **long**\* a = NULL;
11. #pragma omp parallel private(i)
12. {**int** id = omp\_get\_thread\_num();
13. **for**(i=id;i<N;i+=NUM\_THREADS)
14. arr[i] = i+1;
15. }
16. a = (**long** **long** \*)malloc(**sizeof**(**long** **long**)\*NUM\_THREADS);
17. **long** **long** sum = 0;
18. **double** k = 0;
19. start = omp\_get\_wtime();
20. #pragma omp parallel private(i)
21. {**int** id = omp\_get\_thread\_num();
22. **for**(i=id,a[id]=0;i<N-2;i+=NUM\_THREADS){
23. a[id] += (arr[i]+arr[i+1]+arr[i+2]);
24. }
25. }
26. **for**(i=0;i<NUM\_THREADS;i++){
27. sum += a[i];
28. }
29. k = sum/2.0;
30. free(a);
31. a=NULL;
32. end = omp\_get\_wtime();
33. printf("%d\t%lf\t%f\n",NUM\_THREADS,k,end-start);
34. **return** 0;
35. }

②计算数列和 paralle for

1. #include <stdio.h>
2. #include <omp.h>
3. #define N 100000000
4. **int** arr[N];
5. **int** main() {
6. **double** start, end;
7. **int** NUM\_THREADS = omp\_get\_max\_threads();
8. **int** i,j = 0;
9. **double** k =0;
10. #pragma omp prallel for private(i)
11. **for**(i=0; i<N; i++)
12. arr[i] = i+1;
13. **long** **long** sum = 0;
14. start = omp\_get\_wtime();
15. #pragma omp parallel for private(i) reduction(+:sum)
16. **for**(i=0; i<N-2; i++) {
17. sum += (arr[i]+arr[i+1]+arr[i+2]);
18. }
19. k = sum/2.0;
20. end = omp\_get\_wtime();
21. printf("%d\t%f\t%f\n",NUM\_THREADS,k,end-start);
22. **return** 0;
23. }

**实验结果：**

（1），计算pi

①域并行和多线程padding运行效率和加速比

|  |  |  |
| --- | --- | --- |
| **NUM\_THREADS** | **pi\_pad** | **加速比** |
| 1 | 1.67531 | 1 |
| 2 | 0.837921 | 1.999365 |
| 4 | 0.420056 | 3.988302 |
| 6 | 0.281443 | 5.952573 |
| 8 | 0.210454 | 7.960457 |
| 10 | 0.168774 | 9.926351 |
| 20 | 0.085377 | 19.6225 |
| 40 | 0.048925 | 34.24241 |

|  |  |  |
| --- | --- | --- |
| **NUM\_THREADS** | **pi** | **加速比** |
| 1 | 1.67572 | 1 |
| 2 | 0.89416 | 1.874072 |
| 4 | 1.425493 | 1.175537 |
| 6 | 1.010719 | 1.657948 |
| 8 | 0.856088 | 1.957416 |
| 10 | 1.133 | 1.479011 |
| 20 | 0.702226 | 2.386297 |
| 40 | 0.457186 | 3.665292 |

表1：域并行，程序执行时间 表2：使用padding的域并行，程序执行时间

使用padding技术，让有效数据在数组空间中分散存放，可避免因cpu缓存机制导致内存和cpu之间频繁的交换数据，使各个并行线程之间不会有访问内存的冲突，能显著提高并行的效率。

②.表3：不同的多线程方法计算pi值的时间

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **NUM\_THREADS** | **pi** | **pi\_pad** | **pi\_atom** | **pi\_crit** | **pi\_for** |
| 1 | 1.67572 | 1.67531 | 1.675994 | 1.676911 | 1.676476 |
| 2 | 0.89416 | 0.837921 | 0.837964 | 0.838159 | 0.838301 |
| 4 | 1.425493 | 0.420056 | 0.420917 | 0.420547 | 0.420487 |
| 6 | 1.010719 | 0.281443 | 0.284529 | 0.284072 | 0.28178 |
| 8 | 0.856088 | 0.210454 | 0.212637 | 0.210461 | 0.210325 |
| 10 | 1.133 | 0.168774 | 0.168668 | 0.171851 | 0.168577 |
| 20 | 0.702226 | 0.085377 | 0.08561 | 0.086468 | 0.088862 |
| 40 | 0.457186 | 0.048925 | 0.044699 | 0.044524 | 0.045749 |

上表各列时间从左到右分别表示域并行，使用padding技术的域并行，atomic语句，critic语句，parallel for语句程序执行的时间，使用单纯的域并行，因为数组内存访问冲突，并行效率很低。使用其他四种并行计算方式，程序执行时间相差不大，并行效率都很高。

(2).使用域并行和parallel for计算数列和的时间和加速比

|  |  |  |
| --- | --- | --- |
| **NUM\_THREADS** | **time** | **加速比** |
| 1 | 0.471321 | 1 |
| 2 | 0.915235 | 0.514973 |
| 4 | 0.650384 | 0.724681 |
| 6 | 0.507307 | 0.929065 |
| 8 | 0.682836 | 0.69024 |
| 10 | 0.55562 | 0.848279 |
| 20 | 0.40738 | 1.156957 |
| 40 | 0.36393 | 1.295087 |

|  |  |  |
| --- | --- | --- |
| **NUM\_THREADS** | **time** | **加速比** |
| 1 | 0.340226 | 1 |
| 2 | 0.167395 | 2.032474 |
| 4 | 0.086067 | 3.953037 |
| 6 | 0.056277 | 6.04556 |
| 8 | 0.043712 | 7.783355 |
| 10 | 0.035332 | 9.629401 |
| 20 | 0.035362 | 9.621232 |
| 40 | 0.029725 | 11.44579 |

表4：域并行计算数组和 表5：parallel for 计算数组和

表4是使用域并行计算数组和程序执行时间和加速比，因数列采用一维数组存储，在一个连续的空间内，当不同的线程在取数时，因cpu的缓存机制，会取出目标内存块及附近的内存，同时阻止其他线程对这些内存进行读写，导致了伪共享。所以随着线程数增加，程序执行时间可能反而增加，并行效率很低。