# Assignments of lesson2

Performance statistics of Bucket Sorted Problems

Sequential way:

|  |  |
| --- | --- |
| **Phase** | **Execution Time** |
| Distribution | 81 |
| Sort | 4183 |
| Merge | 298 |
| Sum | 4562 |

Parallel with Single Lock:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Phase** | **Sequence** | **Time** | **Processors** | **Speed Up** | **PU** |
| Distribution | 81 | 1823 | 4 | 0.044432255 | 0.011108064 |
| Sort | 4183 | 2084 | 4 | 2.007197697 | 0.501799424 |
| Merge | 298 | 150 | 4 | 1.986666667 | 0.496666667 |
| Sum | 4562 | 4057 | 4 | 1.124476214 | 0.281119053 |

Parallel with Fine Grain Locks:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Phase** | **Sequence** | **Time** | **Processors** | **Speed Up** | **PU** |
| Distribution | 81 | 359 | 4 | 0.225626741 | 0.056406685 |
| Sort | 4183 | 2132 | 4 | 1.962007505 | 0.490501876 |
| Merge | 298 | 164 | 4 | 1.817073171 | 0.454268293 |
| Sum | 4562 | 2655 | 4 | 1.71826742 | 0.429566855 |

Questions:

The reason for more executing time used for distribution phase in parallel solution with single lock is because the multiple threads have to wait for the shared data to be unlocked. As for process concerning with the shared data, it’s actually sequential for different threads. So it would take more time than the sequential solution.

Code:

void sequential() {

//Init the list;

for (int i = 0; i < n; i++) {

list[i] = rand();

}

for (int i = 0; i < m; i++) {

for (int j = 0; j < bsize; j++) {

bucket[i][j] = 0;

}

}

for (int i = 0; i < m; i++) {

count[i] = 0;

}

clock\_t t = clock();

//Distribute the numbers

for (int i = 0; i < n; i++) {

int bnum = (int)(((long double)m) \* ((((long double)list[i])) / ((long double)range)));

bucket[bnum][count[bnum]++] = list[i];

}

printf("Time used for distribution data: %d\n", clock() - t);

clock\_t timeForSort = clock();

//Quick sort all the buckets

for (int i = 0; i < m; i++) {

qsort(bucket[i], count[i], sizeof(int), lt);

}

printf("Time used for sort: %d\n", clock() - timeForSort);

//Merge all the data from the buckets

clock\_t timeForMerge = clock();

int finalIndex = 0;

for (int i = 0; i < m; i++) {

for (int j = 0; j < bsize; j++) {

if (bucket[i][j] != 0) {

final[finalIndex++] = bucket[i][j];

}

}

}

printf("Time used for merge: %d\n", clock() - timeForMerge);

printf("Time used: %d\n", clock() - t);

/\*\*Sequential time:

Time used for distribution data : 91

Time used for sort : 4210

Time used for merge : 291

Time used totally: 4870 \*/

}

void parallelWithSingleLock() {

omp\_lock\_t my\_lock;

omp\_init\_lock(& my\_lock);

//Init the list;

for (int i = 0; i < n; i++) {

list[i] = rand();

}

for (int i = 0; i < m; i++) {

for (int j = 0; j < bsize; j++) {

bucket[i][j] = 0;

}

}

for (int i = 0; i < m; i++) {

count[i] = 0;

}

clock\_t t = clock();

//Distribute the numbers

int threadNum = omp\_get\_num\_procs();

omp\_set\_num\_threads(threadNum);

#pragma omp parallel for

for (int i = 0; i < n; i++) {

int bnum = (int)(((long double) m) \* ((((long double)list[i])) / ((long double)range)));

//Use single lock

omp\_set\_lock(&my\_lock);

bucket[bnum][count[bnum]++] = list[i];

omp\_unset\_lock(&my\_lock);

}

printf("Time used for distribution data: %d\n", clock() - t);

clock\_t timeForSort = clock();

//Quick sort all the buckets

omp\_set\_num\_threads(threadNum);

#pragma omp parallel for

for (int i = 0; i < m; i++) {

qsort(bucket[i], count[i], sizeof(int), lt);

}

printf("Time used for sort: %d\n", clock() - timeForSort);

clock\_t timeForMerge = clock();

//Merge all the data from the buckets

omp\_set\_num\_threads(threadNum);

int finalIndex = 0;

#pragma omp parallel for

for (int i = 0; i < m; i++) {

for (int j = 0; j < bsize; j++) {

if (bucket[i][j] != 0) {

final[finalIndex++] = bucket[i][j];

}

}

}

printf("Time used for merge: %d\n", clock() - timeForMerge);

/\*\*for (int i = 0; i < n; i++) {

printf("%d\n", final[i]);

}\*/

printf("Total Time used: %d", clock() - t);

}

void parallelWithFineGrainedLock() {

omp\_lock\_t my\_lock[m];

for (int i = 0; i < m; i++) {

omp\_init\_lock(&my\_lock[i]);

}

//Init the list;

for (int i = 0; i < n; i++) {

list[i] = rand();

}

for (int i = 0; i < m; i++) {

for (int j = 0; j < bsize; j++) {

bucket[i][j] = 0;

}

}

for (int i = 0; i < m; i++) {

count[i] = 0;

}

clock\_t t = clock();

//Distribute the numbers

int threadNum = omp\_get\_num\_procs();

omp\_set\_num\_threads(threadNum);

#pragma omp parallel for

for (int i = 0; i < n; i++) {

int bnum = (int)(((long double)m) \* ((((long double)list[i])) / ((long double)range)));

//Use single lock

omp\_set\_lock(&my\_lock[bnum]);

bucket[bnum][count[bnum]++] = list[i];

omp\_unset\_lock(&my\_lock[bnum]);

}

printf("Time used for distribution data: %d\n", clock() - t);

clock\_t timeForSort = clock();

//Quick sort all the buckets

omp\_set\_num\_threads(threadNum);

#pragma omp parallel for

for (int i = 0; i < m; i++) {

qsort(bucket[i], count[i], sizeof(int), lt);

}

printf("Time used for sort: %d\n", clock() - timeForSort);

clock\_t timeForMerge = clock();

//Merge all the data from the buckets

omp\_set\_num\_threads(threadNum);

int finalIndex = 0;

#pragma omp parallel for

for (int i = 0; i < m; i++) {

for (int j = 0; j < bsize; j++) {

if (bucket[i][j] != 0) {

final[finalIndex++] = bucket[i][j];

}

}

}

printf("Time used for merge: %d\n", clock() - timeForMerge);

/\*\*for (int i = 0; i < n; i++) {

printf("%d\n", final[i]);

}\*/

printf("Total Time used: %d", clock() - t);

/\*\*

Time used for distribution data : 359

Time used for sort : 2132

Time used for merge : 164

Total Time used : 2656 \*/

}