ISSN:0975-9646

Performance Analysis of Counting Sort Algorithm using various Parallel Programming Models

M Rajasekhara Babu, M Khalid, Sachin Soni, Sunil Chowdari Babu, Mahesh

School of Computing Science and Engineering
VIT University, Vellore, India.

Abstract: The olden ways of programming does not utilize the advantage of multi-core systems. In order to fully exploit these multi-core machines, organizations need to redesign applications so that the processors can treat them as multiple threads of execution. Programmers need to hunt for optimum spots in their codes to insert the parallel code, divide the work approximately into equal parts that can be run simultaneously and associate the precise times for the communication of the threads. Redesigning applications to implement recognition of the core speed of one core by another core in the die must also be taken into grave consideration. As Jones points out, "While that next-generation chip will have more CPUs, each individual CPU will be no faster than the previous year's model. If we want our programs to run faster, we must learn to write parallel programs". Therefore, software developers must take steps to modify the traditional way of writing programs to make way for the implementation of concurrency". Parallelism is strategy for performing complex and large programs faster. The large tasks can be decomposed in to smaller tasks and execute simultaneously.

Keywords: Multi-core architecture, Parallel Programming, Multiple Threads, OpenMP (API), MPI, Concurrent JAVA.

1. INTRODUCTION

Counting sort sorts the values over specific range. It counts the number of occurrences of each value and then calculates the number of values less than each value. Then it places the values in sorted order based on the count of the values. If there are X values less than Y then place the place the Y value in Xth position. Let N be the number of values in an array and K be the range of values present in the array then the time complexity of this algorithm in an average case and worst case is O (N+K). We can reduce the execution time by identifying the concurrent code in the algorithm and executing it parallel. We can examine this concurrent code using Open MP, MPI and concurrent java.

2. COUNTING SORT

Algorithm for Counting Sort

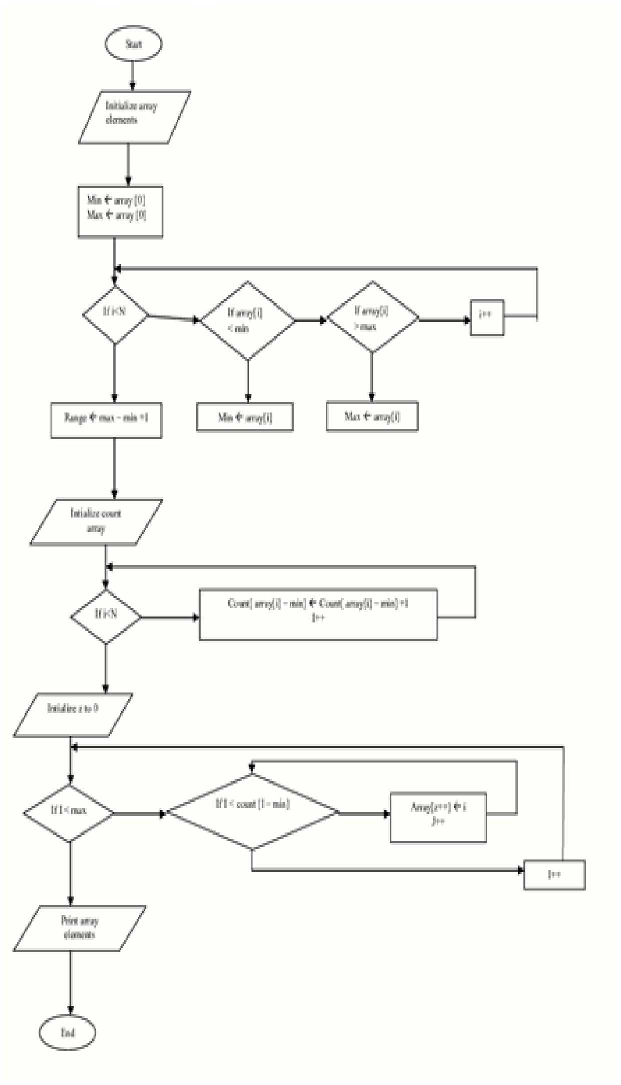
```
Input: Size of the array
Output: Sorted array and its Execution time
Method:
Begin
1.      Initialize array[N]
2.      Min ← array[0]
3.      Max ← array[0]
4.      For i ← 1 to N
Do
5.      If array[i] < min
6.      Min ← array[i]
7.      If array[i] > max
8.      Max ← array[i]
end For
9.      Range ← max – min + 1
10.     Initialize count[range+1]
11.     For i ← 0 to N
Do
12.     count[ array[i] - min ] ← count[ array[i] - min ] + 1
end for
13.     initialize z ← 0
14.     for i ← min to max
do
15.     for j ← 0 to count[ i - min ]
do
16.     array[z++] ← i
end for
end for
17.     for i ← 0 to N-1
do
18.     print "array[i]"
end for
End
```

In this algorithm first we determine the minimum and maximum values of array in order to find the range of an array. Using that range value we initialize another array count of size range+1 to zero. We calculate the elements of count array by considering the actual array elements, suppose the value of array is 4 then the 5th location value of count array is 1, like this we find all values of count array. Then we calculate number of elements present before each value of count array and save the count in the count array itself. With the help of count array rearrange the original array elements. It becomes sorted list. So such a way no where we have used nested for loop so advantage of this algorithm is it's time complexity is θ(n), where n is no of element of the array, but as far as concern the disadvantage of this algorithm is more space complex and if any one element that value is max size it have to create that much long length of array, so it will perform less if input stream in following manner,

2	4	5	989	7	8	9	0

For this type of array we have to maintain a long size of array which index 0 to 989.

3. FLOW CHART:



4. METHODOLOGY USED FOR PARALLELIZE WITH DIFFERENT PARALLEL PROGRAMMING MODELS

OpenMP:

As the algorithm implementation shows the main dominating and time consuming for loop is following

```
For (i=0;i<size;i++)

{

Count [array1 [i]-min] ++;

}
```

To parallelize this for loop we have two mythology in OpenMP following one with partition in to SECTIONS and second one parallelize the FOR loop.

Parallelizing_using_sections

If we are taking in to the *SECTIONS* then we have core 2 due processor and we can make at max 2 sections to make the result efficiently. So make a partition in the FOR loop also because both of the cores will take care of half of the *count [array[i]]*.

Following code fragment used

```
#pragma omp parallel sections{

#pragma omp section{

For (i=0;i<size/2;i++)

Count [array1[i]-min]++;}

#pragma omp section{

For (i=size/2; i<size; i++)

Count [array1 [i]-min] ++;}

  }
```

Parallelize with parallel for

In this inter loop dependency when calculating the count [] elements so reduction () DATA SHARING ATTRIBUTE CLAUSE of *FOR* loop is used, in following manner

```
#pragma omp parallel for reduction (+:count)  {

For (i=0;i<size;i++)

Count [array[i]-min]= Count [array[i]-min]+1;}
```

MPI:

If we compare with OpenMP here processes will execute of each of the processor one processes. So if we are comparing with core2due processor so we have taken only two processes with 2 cores. So we have used counting sort function for each of the processes after partition the data or array then after one loop Marge (agglomeration) both of output array. The following manner used

```
If (id==0){

count_sort(a*,0,size/2);

}


 If(id==1){

count_sort(a*,size/2,size);

}

If (id==0){

agglomeration_marge(a*,b*,0,size/2,size);

}
```

Implementation with concurrent JAVA:

Same as OpenMP JAVA parallelism works on threads executions then a pool of threads is created in JAVA and as a parameter we have passed 2 because for getting a efficient performance we have core2 due processors.

Executor pool=Executors.newFixedThreadPool(2);

This is main dominating for creating treads and executed the function count_sort().

5. RESULTS:

OpenMP :

Array size	Serial program readings	Parallel program readings
10	0.000011	0.000008
100	0.000017	0.000011
1000	0.000089	0.000059

Table: 5.1 OpenMP readings

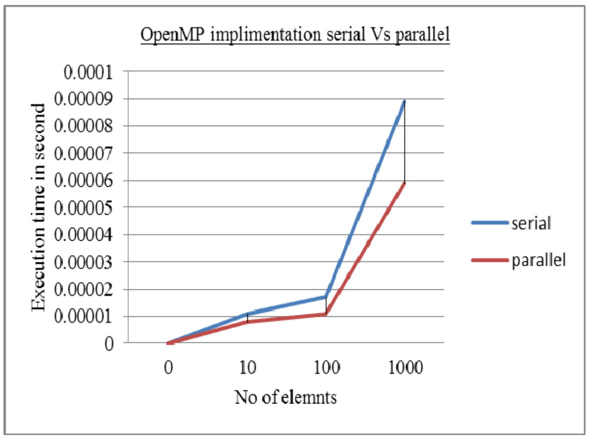


Fig: 5.1 OpenMP Graph

So at the beginning the difference is not that much significant in the graph and when the values of array taken higher then significant difference is observed, and speed up is in band of 1.55 to 1.35 which is more accurate.

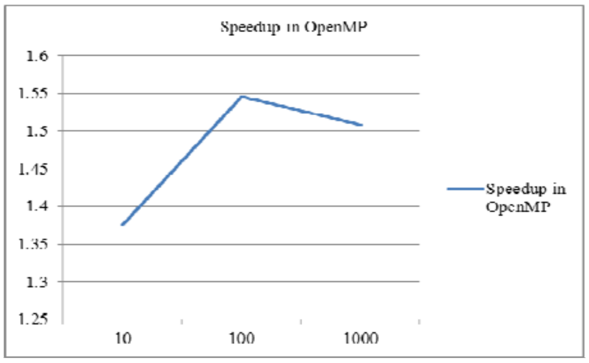


Fig 5.2 Open MP Speedup Graph

MPI:

Table 5.2 MPI readings

Array size	Serial program readings	Parallel program readings
10	0.000041	0.00006198
100	0.0006589	0.00007286
1000	0.0001358	0.0001023
10000	0.0007799	0.0004708
100000	0.00725	0.0041

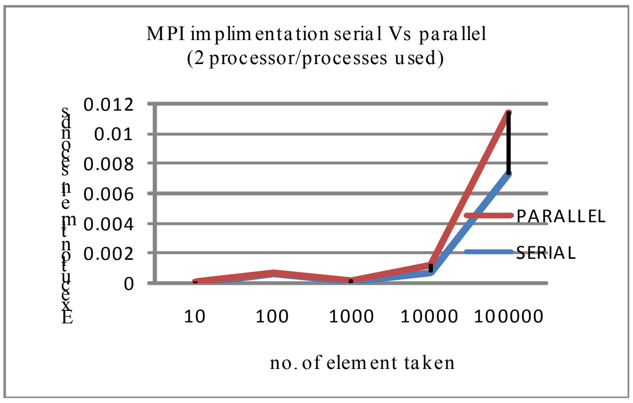


Fig: 5.3 MPI Graph

So at the beginning the difference is not that much significant in the graph and when the values of array taken higher then significant difference is observed, and speed up is in band of 0.80 to 1.80 which is more accurate

SPEEDUP

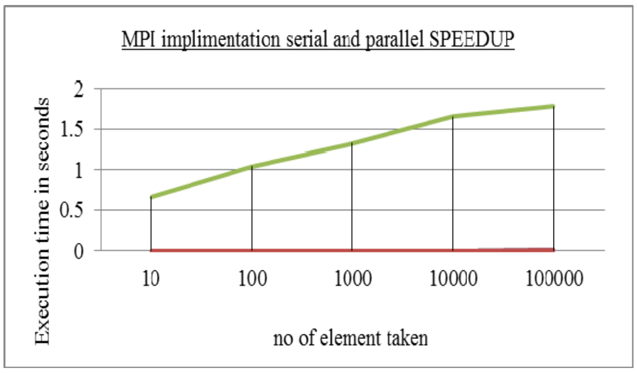


Fig 5.4 MPI Speedup Graph

Concurrent Java:

Table: 5.3 Concurrent java readings

Array size	Serial program execution time	Parallel program execution time
100	3.79	19.54
500	3.8	21.81
1000	20.64	23.47
5000	30	28.42
10000	103	28.87
15000	236	36.86
50000	320	48.85

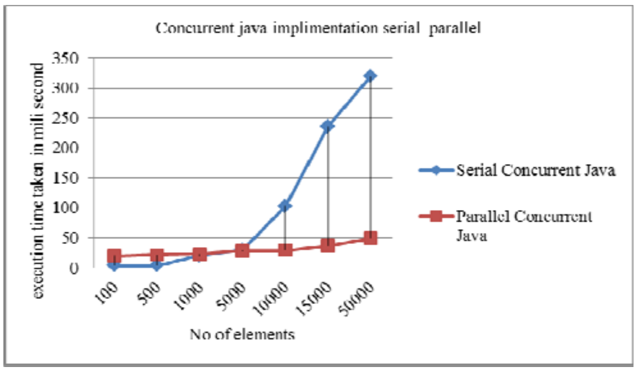


Fig: 5.5 Concurrent java graph

So at the beginning the difference is not that much significant in the graph and when the values of array taken higher then significant difference is observed, and speed up is discrete.

SPEEDUP

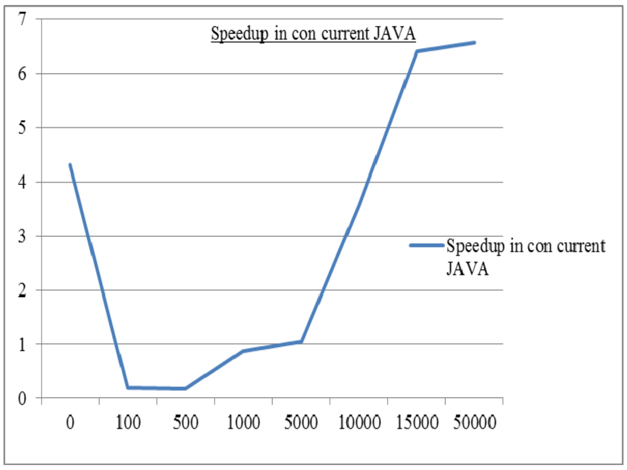


Fig 5.6 Concurrent Java Speedup Graph

6. CONCLUSION

As this report is compared the counting sort with taken different values. In MPI we have 72 core cluster but as comparing result with core2due architecture we have taken only 2 processor / processers. With comparing the result OpenMP is performing well rather than MPI and concurrent java, but for more illustrating MPI and OpenMP for 2 core are performing almost same executing time while concurrent JAVA is getting higher execution time.

REFERENCES

1. www.cse.iitk.ac.in/teaching/courses/CS210.html.
2. Ratnayake, K.; Amer, A.;Concordia Univ., Montreal "An FPGA Architecture of counting-Sorting on a Large Data Volume : Application to Video Signals" Issue 2007 On page(s): 431 – 436.
3 .www.openMP.org
4. www.llnl.gov/computing/tutorials/mpi
5. www. llnl.gov/computing/tutorials/openMP.
6. Podcast from JavaPolis - 'Java Concurrency Utilities in JDK 5.0 by Brian Goetz.
7. Java Concurrency Tutorial by Jakob Jenkov.
8. Thread Safe Java Programming by Vadym Ustymenko.
9. Book "Parallel programming in C with MPI and OpenMP" by Michael J. Quinn.