On the Practical Use of Bead Sort

Dominik Schultes

25. March 2004

Abstract

We discuss some issues regarding the practical use of *Bead Sort* [ACD02]. We do not want to query the basic idea of sorting using a natural algorithm nor the interesting theoretical results, but we concentrate on the aspects that probably prevent that Bead Sort will be very successful in practice. These aspects contain the time and space complexity and the problem of sorting keys that are assigned to data records. Particularly, we want to compare Bead Sort with well known sorting algorithms, especially with Distribution Counting.

1 Introduction

In the Bead Sort algorithm [ACD02] each integer x is represented by x beads that are arranged in a horizontal row on vertical rods. When the input is given in this way, the beads fall down until they hit the bottom or another bead that already rests. If each row with x beads is again interpreted as the number x, the rows represent the sorted data after all beads have reached their final position.

In Section 3, we will compare Bead Sort with the well known Merge Sort [Sed88] and Distribution Counting [Sed88] algorithms.

2 Sorting Keys Linked to Data Records

Practically, in every application not only numbers have to be sorted, but keys that are linked to data records. For instance, in a database, whole records consisting of first name, surname and address are sorted by surname. Another example is sorting as a part of another algorithm, e.g., Kruskal's algorithm to compute a minimum spanning tree in a graph. The first step is sorting the edges by weight, but we are not only interested in the sorted weights, on the contrary, we do not care for the actual weights, but we just want to know the ascending order of the edges. Hence, it is very important in practice to ensure that the links between the keys and the corresponding data are not destroyed by sorting.

Unfortunately, it is in the nature of Bead Sort to lose this information. One possibility to bypass this problem is to use a hash table. If all keys are distinct, you can store a link to the data in the hash table according to the key. If there are multiple records with the same key, you can use a linked list at the corresponding position in the hash table. After filling the hash table you can apply Bead Sort and when the result is read, you can access the hash table in order to retrieve the data that is linked to the sorted keys. But, of course, then the whole procedure is no natural algorithm anymore.

3 Time Complexity

Depending on the point of view, [ACD02] and [Aru04] present different time complexities for Bead Sort. One point of view is the interpretation of falling beads that are accelerated by gravity and therefore need a time that depends only on the square root of the height. We consider this interpretation as very fascinating because not only the lower bound for comparison based sorting of $O(n \log n)$ is beaten, but also $O(n)$. However, we doubt that this is useful in practice as there is already a lower bound of $O(n)$ for the input. Thus, the time we need to sort n numbers depends on $O(n)$ anyway.

Let us compare on a conventional machine a simple sequential implementation of Bead Sort with Merge Sort and with Distribution Counting. We denote the number of keys with n and the biggest key with m. A sequential implementation of Bead Sort has the time complexity $O(n \cdot m)$, Merge Sort $O(n \log n)$, and Distribution Counting $O(n + m)$.

For a small m, Bead Sort can beat Merge Sort, but for bigger keys (i.e., a larger m) and a reasonable number of keys, Bead Sort gets very slow in comparison to Merge Sort. Theoretically, Bead Sort is asymptotically faster than Merge Sort for any fixed m, but in practice, the memory is exceeded before n is so big that Bead Sort

can overtake Merge Sort.

Since $O(n + m)$ is always better than $O(n \cdot m)$, Distribution Counting is throughout faster than Bead Sort. Figure 1 and 2 represent the results of the measurements that we have performed with the help of a C++ program (see Appendix A). The source code and the results of the measurements are also available at [Sch04].

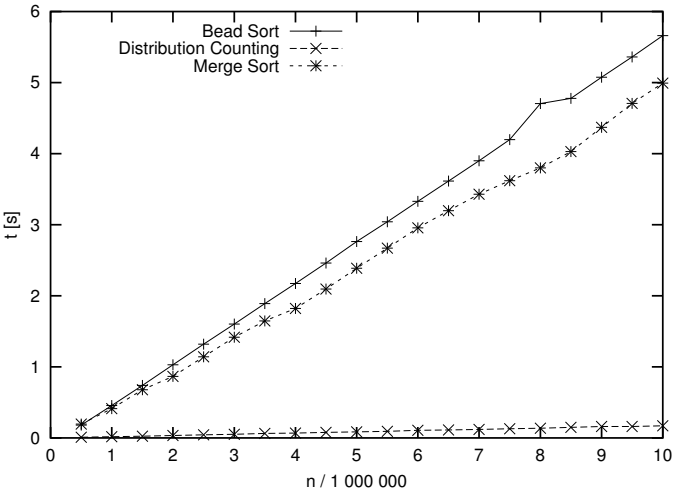

Figure 1: Comparison between Bead Sort, Distribution Counting and Merge Sort.

The number of keys n varies between 500 000 and 10 000 000 with a step size of 500 000, no key is bigger than a fixed $m = 100$. The measurements are done on a Intel Centrino 1.5 GHz using Linux 2.4.19 and the g++ compiler 3.2 with optimization level 6 (-O6).

Of course, Bead Sort has not been designed with a simple sequential implementation on a conventional machine in mind, but it should take advantage of parallelization. In [ACD02] three different implementations are introduced basing on analog hardware, on a cellular automaton resp. on digital hardware. These implementations have in common that they reduce the time complexity by parallelization from $O(n \cdot m)$ to $O(n)$. However, we should keep in mind that Distribution Counting has – even without parallelization – a quite similar complexity (especially with the assumption that $m \leq n$). Furthermore, we have to admit that the parallelization of Bead Sort does not come for free, which leads to the next issue.

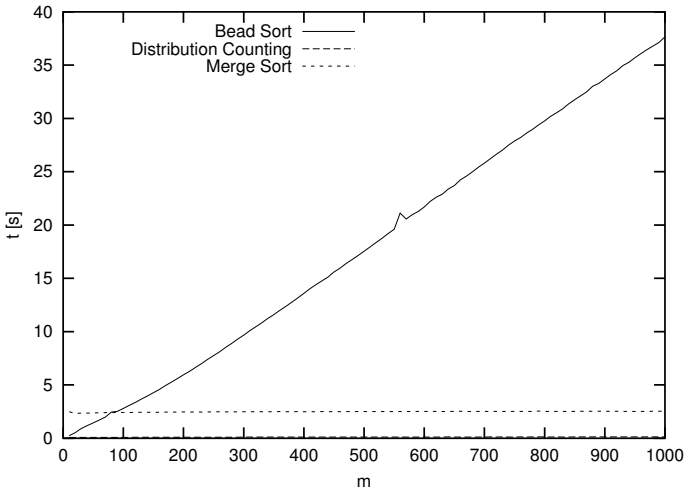

Figure 2: Comparison between Bead Sort, Distribution Counting and Merge Sort.

The number of keys n is fixed to 5 000 000, the maximum key m varies between 10 and 1000 with a step size of 10. Note that Distribution Counting is so fast that the corresponding line is hardly visible. Obviously, the value for Bead Sort for $m = 560$ is a measurement error.

4 Space Complexity

The space complexity of Bead Sort depends on m and, in contrast to the time complexity, this cannot be improved. This means that in general the required space grows exponentially with the input length – regardless of the chosen implementation. Hence, Bead Sort can only be used for an input with a small, fixed m. (This restriction applies to Distribution Counting as well.)

5 Conclusion

While Bead Sort is a fascinating natural algorithm with interesting theoretical aspects, it probably will not succeed in practice. Due to its space complexity it cannot be applied to input data with large keys and for small keys, the linear time Distribution Counting algorithm is very competitive. Furthermore, the natural Bead Sort algorithm tends to destroy the links between keys and data records.

References

[ACD02] J. J. Arulanandham, C. S. Calude, and M. J. Dinneen. Bead-sort: A natural sorting algo-

rithm. *Bulletin of the European Association for Theoretical Computer Science*, 76:153–162, 2002.

[Aru04] J. J. Arulanandham. Bead-sort – a natural algorithm for sorting. http://www.cs.auckland.ac.nz/∼cristian/umc/bead_sort.zip, March 2004.

[Sch04] Dominik Schultes. A comparison between bead sort, distribution counting and merge sort. http://www-user.rhrk.uni-kl.de/∼dschult/umc/asg2/, March 2004.

[Sed88] R. Sedgewick. *Algorithms*. Addison-Wesley, 2nd edition, 1988.

A Source Code

```
/*****************************
 * A Comparison between BEAD SORT, DISTRIBUTION COUNTING
 * and MERGE SORT.
 * by Dominik Schultes
 * 25. March 2004
 *****************************/

#include <sys/time.h>
#include <iostream>
#include <fstream>
#include <vector>


using namespace std;

typedef vector<int> my_vector;


/*****************************
 * This section contains methods to measure the used time
 * and to log the measured values.
 *****************************/
bool log_n = true;
double last_timestamp;

/** Returns a current timestamp. */
inline double timestamp()
{
    struct timeval tp;
    gettimeofday (&tp, NULL);
    return double (tp.tv_sec) + tp.tv_usec / 1000000.;
}

/** Starts the time measurement. */
inline void start_timer() {
    last_timestamp = timestamp();
}

/** Stops the time measurement and returns the elapsed time. */
inline double stop_timer() {
    return timestamp() - last_timestamp;
}

/**
   Writes the given time to the given output stream.
   Depending on the global variable log_n either n or m
   is used as variable that the given time is assigned to.
*/
inline void logTime(int m, int n, double t, ostream &out) {
    if (log_n) out << n; else out << m;
    out << " " << t << endl;
}


/*****************************
 * This section contains methods to generate the input
 * randomly and to check the output if it is really sorted
 *****************************/

/**
   Generates n random numbers between 1 and m and inserts them
   to the given vector a.
*/
void generateData(my_vector &a, int n, int m) {
    for (int i=0; i<n; i++) {
        a.push_back( (int)(rand() / (double)(RAND_MAX+1.0) * m)+1 );
    }
}

/**
   Checks if the given vector a is sorted in descending order.
   Writes a message to standard out.
*/
void checkData(my_vector &a) {
    for (int i=1; i<a.size(); i++) {
        if (a[i] > a[i-1]) {
            cout << "Check FAILED !" << endl << (i-1) << ": "
                 << a[i-1] << endl << i << ": " << a[i] << endl;
            return;
        }
    }
    cout << "Check passed. Data is sorted." << endl;
}


/*****************************
 * This section contains the actual sorting methods:
 * Bead Sort, Distribution Counting and Merge Sort
 *****************************/

/** A simple implementation of Bead Sort. */
void beadSort(my_vector &a, int m, ostream &out) {
    // initialize
    int n = a.size();

    int *level_count = new int[n+1];
    int *rod_count = new int[m+1];

    for (int i=1; i<=n; i++) level_count[i] = 0;
    for (int i=1; i<=m; i++) rod_count[i] = 0;

    // sort
    start_timer();

    for (int i=0; i<n; i++) {
        for (int j=1; j<=a[i]; j++) {
            ++level_count[ ++rod_count[j] ];
```

```
        }
    }

    logTime( m, n, stop_timer(), out );

    // write sorted data back and clean up
    a.clear();
    for (int i=1; i<=n; i++) a.push_back( level_count[i] );

    delete level_count;
    delete rod_count;
}

/** A simple implementation of Distribution Counting. */
void distributionCounting(my_vector &a, int m, ostream &out) {
    // initialize
    int n = a.size();

    int *result = new int[n];
    int *buckets = new int[m+1];
    for (int i=1; i<=m; i++) buckets[i] = 0;

    // sort
    start_timer();

    for (int i=0; i<n; i++) buckets[a[i]]++;

    for (int i=m-1; i>=1; i--) buckets[i] += buckets[i+1];

    for (int i=0; i<n; i++) result[ --buckets[a[i]] ] = a[i];

    logTime( m, n, stop_timer(), out );

    // write sorted data back and clean up
    a.clear();
    for (int i=0; i<n; i++) a.push_back( result[i] );

    delete result;
    delete buckets;
}

/** A recursive implementation of Merge Sort. */
void mergeSortRecursion(my_vector &a, int l, int r) {
    // base cases
    if (l == r) return;
    if (r - l == 1) {
        if (a[r] > a[l]) {
            int tmp = a[l]; a[l] = a[r]; a[r] = tmp;
        }
        return;
    }

    // recursive calls
    int m = (r-l)/2 + l;
    mergeSortRecursion(a,l,m);
    mergeSortRecursion(a,m+1,r);

    // merge
    my_vector b;
    b.reserve(r-l+1);
    int j = l; int k = m+1;
    for (int i=l; i<=r; i++) {
        if ((j > m) || ((k <= r) && (a[j] < a[k])))
            b.push_back(a[k++]);
        else
            b.push_back(a[j++]);
    }

    // write sorted data back
    j = 0;
    for (int i=l; i<=r; i++) a[i] = b[j++];
}

/**
   Invokes the recursive Merge Sort with the appropriate
   arguments.
*/
void mergeSort(my_vector &a, int m, ostream &out) {
    start_timer();
    mergeSortRecursion(a, 0, a.size()-1);
    logTime( m, a.size(), stop_timer(), out );
}


/*****************************
 * This section contains the main method and methods that
 * control the test runs.
 *****************************/

/**
   Performs several test runs. The number of keys n varies between
   500 000 and 10 000 000 with a step size of 500 000. No key is
   bigger than a fixed m = 100.
*/
void measurement1() {
    ofstream outFileDistrCount( "distrCount1.dat" );
    ofstream outFileBead( "beadSort1.dat" );
    ofstream outFileMerge( "mergeSort1.dat" );

    int m = 100;
    log_n = true;
    int step = 500000;

    for (int n=step; n<=20*step; n+=step) {
        cout << n << endl;

        // generate the data
        my_vector *data1 = new my_vector;
        generateData(*data1, n, m);
        my_vector *data2 = new my_vector(*data1);
        my_vector *data3 = new my_vector(*data1);

        // sort
        distributionCounting(*data1,m,outFileDistrCount);
            beadSort(*data2,m,outFileBead);
        mergeSort(*data3,m,outFileMerge);

        // check the results and clean up
        assert(*data1 == *data2);
        assert(*data1 == *data3);

        checkData(*data1);
        checkData(*data2);
        checkData(*data3);

        delete data1;
        delete data2;
        delete data3;
```

```
    }
}

/**
   Performs several test runs. The number of keys n is fixed
   to 5 000 000. The maximum key m varies between 10 and 1000
   with a step size of 10.
*/
void measurement2() {
    ofstream outFileDistrCount( "distrCount2.dat" );
    ofstream outFileBead( "beadSort2.dat" );
    ofstream outFileMerge( "mergeSort2.dat" );

    int n = 5000000;
    log_n = false;

    for (int m=10; m<=1000; m+=10) {
        cout << m << endl;

        // generate the data
        my_vector *data1 = new my_vector;
        generateData(*data1, n, m);
        my_vector *data2 = new my_vector(*data1);
        my_vector *data3 = new my_vector(*data1);

        // sort
        distributionCounting(*data1,m,outFileDistrCount);
            beadSort(*data2,m,outFileBead);
        mergeSort(*data3,m,outFileMerge);

        // check the results and clean up
        assert(*data1 == *data2);
        assert(*data1 == *data3);

        checkData(*data1);
        checkData(*data2);
        checkData(*data3);

        delete data1;
        delete data2;
        delete data3;
    }
}

/** The main method. */
int main() {

    measurement1();
    measurement2();

    return 0;
}
```