Spreadsort

Spreadsort is a sorting algorithm invented by Steven J. Ross in 2002.[1] It combines concepts from distribution-based sorts, such as radix sort and bucket sort, with partitioning concepts from comparison sorts such as quicksort and mergesort. In experimental results it was shown to be highly efficient, often outperforming traditional algorithms such as quicksort, particularly on distributions exhibiting structure and string sorting. There is an open-source implementation with performance analysis and benchmarks[2], and HTML documentation [3].

Quicksort identifies a *pivot element* in the list and then partitions the list into two sublists, those elements less than the pivot and those greater than the pivot. Spreadsort generalizes this idea by partitioning the list into *n*/*c* partitions at each step, where *n* is the total number of elements in the list and *c* is a small constant (in practice usually between 4 and 8 when comparisons are slow, or much larger in situations where they are fast). It uses distribution-based techniques to accomplish this, first locating the minimum and maximum value in the list, and then dividing the region between them into *n*/*c* equal-sized bins. Where caching is an issue, it can help to have a maximum number of bins in each recursive division step, causing this division process to take multiple steps. Though this causes more iterations, it reduces cache misses and can make the algorithm run faster overall.

In the case where the number of bins is at least the number of elements, spreadsort degenerates to bucket sort and the sort completes. Otherwise, each bin is sorted recursively. The algorithm uses heuristic tests to determine whether each bin would be more efficiently sorted by spreadsort or some other classical sort algorithm, then recursively sorts the bin.

Like other distribution-based sorts, spreadsort has the weakness that the programmer is required to provide a means of converting each element into a numeric key, for the purpose of identifying which bin it falls in. Although it is possible to do this for arbitrary-length elements such as strings by considering each element to be followed by an infinite number of minimum values, and indeed for any datatype possessing a total order, this can be more difficult to implement correctly than a simple comparison function, especially on complex structures. Poor implementation of this *value* function can result in clustering that harms the algorithm's relative performance.

Contents

Performance

The worst-case performance of spreadsort is O(n log n) for small data sets, as it uses introsort as a fallback. In the case of distributions where the size of the key in bits k times 2 is roughly the square of the log of the list size n or smaller ($2k < (\log n)^2$), it does better in the worst case, achieving $O(n \sqrt{k - \log n})$ worst-case time for the originally published version, and $O(n \cdot ((k/s) + s))$ for the cache aware version. For many real sorting problems with over 1000 items, including string sorting, this asymptotic worst-case is better than O(n log n).

Experiments were done comparing an optimized version of spreadsort to the highly optimized C++ `std::sort`, implemented with introsort. On lists of integers and floats spreadsort shows a roughly 2–7× runtime improvement for random data on various operating systems.[1] (http://www.boostpro.com/vault/index.php?action=downloadfile&filename=algorithm_sorting.zip&directory=&)

In space performance, spreadsort is worse than most in-place algorithms: in its simplest form, it is not an in-place algorithm, using O(n) extra space; in experiments, about 20% more than quicksort using a c of 4–8. With a cache-aware form (as included in Boost.Sort), less memory is used and there is an upper bound on memory usage of the maximum bin count times the maximum number of recursions, which ends up being a few kilobytes times the size of the key in bytes. Although it uses asymptotically more space than the O(log n) overhead of quicksort or the O(1) overhead of heapsort, it uses considerably less space than the basic form of mergesort, which uses auxiliary space equal to the space occupied by the list.

Implementation

```
unsigned
RoughLog2(DATATYPE input)
{
    unsigned char cResult = 0;
    // The && is necessary on some compilers to avoid infinite loops; it doesn't
    // significantly impair performance
    if(input >= 0)
        while((input >> cResult) && (cResult < DATA_SIZE)) cResult++;
    else
        while(((input >> cResult) < -1) && (cResult < DATA_SIZE)) cResult++;
    return cResult;
}
SIZETYPE
GetMaxCount(unsigned logRange, unsigned uCount)
{
    unsigned logSize = RoughLog2Size(uCount);
    unsigned uRelativeWidth = (LOG_CONST * logRange)/((logSize > MAX_SPLITS) ? MAX_SPLITS :
logSize);
    // Don't try to bitshift more than the size of an element
    if(DATA_SIZE <= uRelativeWidth)
        uRelativeWidth = DATA_SIZE - 1;
    return 1 << ((uRelativeWidth < (LOG_MEAN_BIN_SIZE + LOG_MIN_SPLIT_COUNT)) ?
        (LOG_MEAN_BIN_SIZE + LOG_MIN_SPLIT_COUNT) :  uRelativeWidth);
}

void
FindExtremes(DATATYPE *Array, SIZETYPE uCount, DATATYPE & piMax, DATATYPE & piMin)
{
    SIZETYPE u;
    piMin = piMax = Array[0];
    for(u = 1; u < uCount; ++u){
        if(Array[u] > piMax)
            piMax=Array[u];
        else if(Array[u] < piMin)
            piMin= Array[u];
    }
}

//---------------------SpreadSort Source-----------------

Bin *
SpreadSortCore(DATATYPE *Array, SIZETYPE uCount, SIZETYPE & uBinCount, DATATYPE &iMax, DATATYPE
```

```
&iMin)
{
    // This step is roughly 10% of runtime but it helps avoid worst-case
    // behavior and improves behavior with real data.  If you know the
    // maximum and minimum ahead of time, you can pass those values in
    // and skip this step for the first iteration
    FindExtremes((DATATYPE *) Array, uCount, iMax, iMin);
    if(iMax == iMin)
        return NULL;
    DATATYPE divMin,divMax;
    SIZETYPE u;
    int LogDivisor;
    Bin * BinArray;
    Bin* CurrentBin;
    unsigned logRange;
    logRange = RoughLog2Size((SIZETYPE)iMax-iMin);
    if((LogDivisor = logRange - RoughLog2Size(uCount) + LOG_MEAN_BIN_SIZE) < 0)
        LogDivisor = 0;
    // The below if statement is only necessary on systems with high memory
    // latency relative to processor speed (most modern processors)
    if((logRange - LogDivisor) > MAX_SPLITS)
        LogDivisor = logRange - MAX_SPLITS;
    divMin = iMin >> LogDivisor;
    divMax = iMax >> LogDivisor;
    uBinCount = divMax - divMin + 1;

    // Allocate the bins and determine their sizes
    BinArray = calloc(uBinCount, sizeof(Bin));
    // Memory allocation failure check and clean return with sorted results
    if(!BinArray) {
        printf("Using std::sort because of memory allocation failure\n");
        std::sort(Array, Array + uCount);
        return NULL;
    }

    // Calculating the size of each bin; this takes roughly 10% of runtime
    for(u = 0; u < uCount; ++u)
        BinArray[(Array[u] >> LogDivisor) - divMin].uCount++;
    // Assign the bin positions
    BinArray[0].CurrentPosition = (DATATYPE *)Array;
    for(u = 0; u < uBinCount - 1; u++) {
        BinArray[u + 1].CurrentPosition = BinArray[u].CurrentPosition + BinArray[u].uCount;
        BinArray[u].uCount = BinArray[u].CurrentPosition - Array;
    }
    BinArray[u].uCount = BinArray[u].CurrentPosition - Array;

    // Swap into place.  This dominates runtime, especially in the swap;
    // std::sort calls are the other main time-user.
    for(u = 0; u < uCount; ++u) {
        for(CurrentBin = BinArray + ((Array[u] >> LogDivisor) - divMin);  (CurrentBin->uCount >
u);
            CurrentBin = BinArray + ((Array[u] >> LogDivisor) - divMin))
                SWAP(Array + u, CurrentBin->CurrentPosition++);
        // Now that we've found the item belonging in this position,
        // increment the bucket count
        if(CurrentBin->CurrentPosition == Array + u)
            ++(CurrentBin->CurrentPosition);
    }

    // If we've bucketsorted, the array is sorted and we should skip recursion
    if(!LogDivisor) {
        free(BinArray);
        return NULL;
    }
    return BinArray;
}

void
SpreadSortBins(DATATYPE *Array, SIZETYPE uCount, SIZETYPE uBinCount, const DATATYPE &iMax
                , const DATATYPE &iMin, Bin * BinArray, SIZETYPE uMaxCount)
{
    SIZETYPE u;
    for(u = 0; u < uBinCount; u++){
        SIZETYPE count = (BinArray[u].CurrentPosition - Array) - BinArray[u].uCount;
        // Don't sort unless there are at least two items to compare
        if(count < 2)
            continue;
        if(count < uMaxCount)
            std::sort(Array + BinArray[u].uCount, BinArray[u].CurrentPosition);
        else
            SpreadSortRec(Array + BinArray[u].uCount, count);
    }
    free(BinArray);
```

```
}

void
SpreadSortRec(DATATYPE *Array, SIZETYPE uCount)
{
    if(uCount < 2)
        return;
    DATATYPE iMax, iMin;
    SIZETYPE uBinCount;
    Bin * BinArray = SpreadSortCore(Array, uCount, uBinCount, iMax, iMin);
    if(!BinArray)
        return;
    SpreadSortBins(Array, uCount, uBinCount, iMax, iMin, BinArray,
                   GetMaxCount(RoughLog2Size((SIZETYPE)iMax-iMin), uCount));
}
```

Two Levels are as Good as Any

An interesting result for algorithms of this general type (splitting based on the radix, then comparison-based sorting) is that they are O(*n*) for any bounded and integrable probability density function.[4] This result can be obtained by forcing Spreadsort to always iterate one more time if the bin size after the first iteration is above some constant value. If the key density function is known to be Riemann integrable and bounded, this modification of Spreadsort can attain some performance improvement over the basic algorithm, and will have better worst-case performance. If this restriction cannot usually be depended on, this change will add a little extra runtime overhead to the algorithm and gain little. Other similar algorithms are Flashsort (which is simpler) and Adaptive Left Radix.[5] Adaptive Left Radix is apparently quite similar, the main difference being recursive behavior, with Spreadsort checking for worst-case situations and using std::sort to avoid performance problems where necessary, and Adaptive Left Radix recursing continuously until done or the data is small enough to use insertion sort.

References

1. Steven J. Ross. The Spreadsort High-performance General-case Sorting Algorithm. *Parallel and Distributed Processing Techniques and Applications*, Volume 3, pp. 1100–1106. Las Vegas Nevada. 2002.
2. "Boost.Sort github repository" (https://github.com/boostorg/sort/tree/master). *boostorg/sort*.
3. "HTML Spreadsort Documentation" (http://www.boost.org/doc/libs/1_65_0/libs/sort/doc/html/index.html). Retrieved 30 August 2017.
4. Tamminen, Markku (March 1985). "Two Levels are as Good as Any". *J. Algorithms*. **6** (1): 138–144. doi:10.1016/0196-6774(85)90024-0 (https://doi.org/10.1016%2F0196-6774%2885%2990024-0).
5. Maus, Arne (2002). *ARL, a faster in-place, cache friendly sorting algorithm* (http://www.nik.no/2002/Maus.pdf) (PDF) (Technical report). CiteSeerX 10.1.1.399.8357 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.399.8357).