Fast **mergesort** implementation based on *half-copying* **merge** algorithm

Cezary Juszczak
Institute for Theoretical Physics
University of Wrocław, Poland

April 18, 2007

Abstract

An efficient implementation of **mergesort** is presented in both *copying* and *in-place* versions. It is based on a fast *half-copying* **merge** algorithm. The presented **mergesort** C++ implementation is only 20% slower than **quicksort** and about 50% faster than the current STL **stable_sort** implementation using only half as much memory. It seems a very good candidate for inclusion in the standard STL implemention.

Introduction

The **mergesort** algorithm is one of the most efficient algorithms for sorting data. It was originally used for sorting large data files but can also be used for sorting in memory arrays and lists. When used for sorting lists it has a constant memory demand, but requires an additional memory buffer of size $O(n)$ when used for sorting arrays. It boasts a guranteed $O(n \log n)$ run time and is stable.

The subject of this paper is to describe a fast **mergesort** implementation such that the memory demand is $n/2$ and the real life speed of the C++ implementation is comparable to a **quicksort** implementation.

1 Mergesort

A typical **mergesort** implementation can be described as follows:

```
mergesort(A)
        allocate(B)
        mergesort(A1)
        mergesort(A2)
        merge(A1,A2,B)
        copy(B,A)
```

where A1 and A2 are left and right halves of the array A.

The pure copying in the last line can be avoided if the **copying_mergesort** is used.

```
mergesort(A)
        allocate(B)
        copying_mergesort(A1,B1)
        copying_mergesort(A2,B2)
        merge(B1,B2,A)
```

The **copying_mergesort** is impelemented by means of the **mergesort**:

```
copying_mergesort(A,B)
        mergesort(A1)
        mergesort(A2)
        merge(A1,A2,B)
```

In the above the array B may be used as the **mergesort** buffer so there is no need for additional memory allocation.

2 *Half copying* **merge**

Note that in both cases the **merge** function merges two sorted halves od one array into one other array which is made clear by the notation:

```
        merge(A1,A2,B)
or
        merge(B1,B2,A)
```

What we mean by a half copying **merge** is the merge which is used as follows:

```
        merge(A1,B2,B)
```

Such use of **merge**, where the right half of the input data coincides with the right half of the target location, does not lead to any problems since the place is freed as the data is copied. It may even be advantageous, as it may turn out that some of the data in B2 do not need to be touched. It also results in the possibility of using less memory.

```
mergesort(table A)
        allocate(B1)
        copying_mergesort(A2,B1)
        copying_mergesort(A1,A2)
        merge(B1,A2,A)
        free(B1)
```

The same idea applied to the **copying_mergesort** leads to the following code:

```
copying_mergesort(A,B)
        copying_mergesort(A2,B2)
        copying_mergesort(A1,A2)
        merge(A2,B2,B)
```

Note that the **copying_mergesort** is implemented in terms of itself nad does not use the **mergesort**. Some care needs to be taken in order to preserve stability of the alogrithm since the **merge** used in **mergesort** should first copy elements from its second argument in case of equality while the **merge** used in **copying_mergesort** should first copy elements from its first argument in case of equality.

3 The C++ implementation

The **merge** function merges the [p1,k1) and [p2,k2) sorted ranges. The pointer k2 coincides with the end of the target range [p,k) meaning that no element in [p2,k2) needs to be moved if all elements from [p1,k1) are smaller than *p2. It is assumed that both ranges are non empty. When the range [p1,k1) becomes exhausted the merging is done because all remaining elements from [p2,k2) are already in place.

```
template < class T >
inline  void
merge (T * p1, T * k1, T * p2, T *k2)
{ T* p=p2 - (k1-p1);     //calculate the beginning of the output
  while(true)
  {if(*p1<=*p2)
     {*p++=*p1++;
      if(p1==k1) return;
      }
   else
     {*p++=*p2++;
      if(p2==k2) break;
       }
  }
  do
    *p++=*p1++;
  while(p1!=k1);
}
```

Based on this algorithm a very effcient copying verion of mergesort may be written sorting the elements form the range `[p,k)` and placing the result in the range **[t,t+ (k-p))**. The right half of the input is sorted (by using recursively the same algorithm) into the right half of the output range. Then the left half of the input is range sorted (also recursively) into the right half of the input range. Then it is merged with our half copying merge with the numbers alresdy present in the output.

```
template < class T >
inline void
```

```
copying_mergesort (T * p, T * k, T * t)
{
  if (k > p + 16)
    {
      T *s = p + ((k - p) >>1);
      copying_mergesort (s, k, t+(s-p));
      copying_mergesort (p, s, s);
      merge (s,s+(s-p),t+(s-p),t+(k-p));
    }
  else
     copying_insertionsort (p, k, t);
}
```

For small arrays the insertionsort is used as it is the fastest sorting alogrithm for small n. This also ensures that the **merge** is never called with empty regions [p1,k1) or [p2,k2).

The **mergesort** implementation calls **copying_mergesort**. It sorts the input range [p,k) 'in place' but requires a buffer to temporarily store $\lfloor n/2 \rfloor$ array elements:

```
template < class T >
inline void
mergesort (T * p, T * k)
{
  if (k > p + 16)
    {
      T *s = p + ((k - p) >> 1);
      T *buff=new T[ k-s ];
      copying_mergesort (s, k, buff);
      copying_mergesort (p, s, k - (s - p));
      mergeR (buff, buff + (k - s), k - (s - p), k);
      delete []buff;
    }
  else
     insertionsort (p, k, t);
}
```

To preserve the stability of mergesort we used above a slightly changed version of `merge` – we call it `mergeR` since it assumes that the range [p2,k2)

was to the left of [k1,p1) in the input and takes this into account when the compared elements turn out to be equal (those from [p2,k2) are first copied to the output).

```
template < class T >
inline  void
mergeR (T * p1, T * k1, T * p2, T *k2)
{ T* p=p2 - (k1-p1);
  while(true)
  {if(*p1<*p2)
     {*p++=*p1++;
      if(p1==k1) return;
      }
   else
     {*p++=*p2++;
      if(p2==k2) break;
       }
  }
  do
    *p++=*p1++;
  while(p1!=k1);
}
```

It is remarkable that this implementation is about 30% faster than the current STL `stable_sort` implementation. Is is only 20% slower than quick sort and runs in quranteed $O(n(log(n))$ time using additional memory buffer to accomodate $n/2$ elements. No unnecessary copying is performed. The good performance may be attributed not only to the simplicity of the algorithm and the avoidance od unnecessary copying but also to a smaller memory consumption compared with the traditional implementation which results in more data fitting into the cache memory.

4 The banchmarking program and the results of the tests

Ths is still missing.