Permutation algorithms in the Copenhagen STL

Steffen Nissen
Department of Computer Science, University of Copenhagen,
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
lukesky@diku.dk

Abstract. In this paper I will describe the algorithms and optimization techniques used to implement the next_permutation and prev_permutation functions in the Copenhagen STL. These two functions use reverse, which is why I will also describe the optimizations used in implementing the reverse and reverse_copy functions.

Furthermore I will discuss several ways of implementing the reverse function on forward access containers.

I will mainly discuss implementation details, instead of algorithm details, since the algorithms I use are all well-known.

1. reverse

The reverse function takes two bidirectional iterators as arguments, and reverses the elements between the two iterators.

1.1 Algorithms

The basic algorithm uses two iterators that travel from the ends towards the center. During this the elements that the iterators point to are swapped.

If you use bidirectional iterators, this is the only algorithm that will work in practice. But if you use random access iterators, you can use the enhanced functionality of these iterators to boost performance.

Mortensen [2001] proposed using instruction level parallelism to boost performance. I tried this and instead of having two iterators, I had four iterators working in two pairs. In theory this should give better instruction level parallelism, but since the main resource in this case was the cache, the benchmarks quickly showed that this was not the right way to go. In [Bojesen 2000] Bojesen shows that even though an algorithm looks like it should be more effective, then it might not be, if the memory hierarchies is not used effectively.

Loop unrolling is a straight forward optimisation that can easily be used on this algorithm. This however has a slight disadvantage on small sequences.

Sequences with less than two elements, does not need to be reversed. Making a special case for this is simple, and it reduces the range in which the algorithm suffers from the overhead of loop unrolling.

CPH STL Report 2001-9, April 2001.

1.2 Benchmarks

In order to give a more precise impression of the performance of the algorithms, I have eight different benchmarks, each showing different aspects of the algorithms.

Half of my benchmarks are run on one long sequence, where a new subsequence is reversed each time. This shows how the algorithm behaves when the sequence to be reversed is not already in the cache. The other half of the benchmarks is run on the same sequence each time, which allows efficient use of the cache.

Half of my benchmarks are run on sequences of unsigned integers, and the other half on sequences of 100 byte objects. Since this is a very memory intensive function, the optimizations should not be as evident on the large objects.

Half of my benchmarks are run on vectors, which use random access iterators. The other half is run on lists, which use bidirectional iterators.

The SGI STL implementation [SGI 2001] does not inline the reverse function, but I do not see any reason for not doing this. Which is why the overhead of calling my functions is slightly smaller.

The unoptimized version of my functions, is basically the same as the optimized but without the use of loop unrolling.

1.2.1 Random access iterators

Since the most optimisation is done on the random access iterator version of the algorithm, the best results are expected for this algorithm.

The reversal of a sequence of unsigned integers, with (Figure 1) and without (Figure 2) the elements already in the cache. Shows clearly that the inlining and loop unrolling pays off.

The reversal of a sequence of large objects, with (Figure 3) and without (Figure 4) the elements already in the cache. Shows no real advantage to any of the algorithms, except for sequences of length shorter than 2, where reversal is not necessary.

1.2.2 Bidirectional iterators

Since there is no major differences between my implementation and the SGI STL implementation [SGI 2001], no difference is expected between the different implementations, except for the small overhead of calling the SGI function.

The four benchmark figures (Figure 5, Figure 6, Figure 7 and Figure 8) shows that there is no major difference between the different implementations.

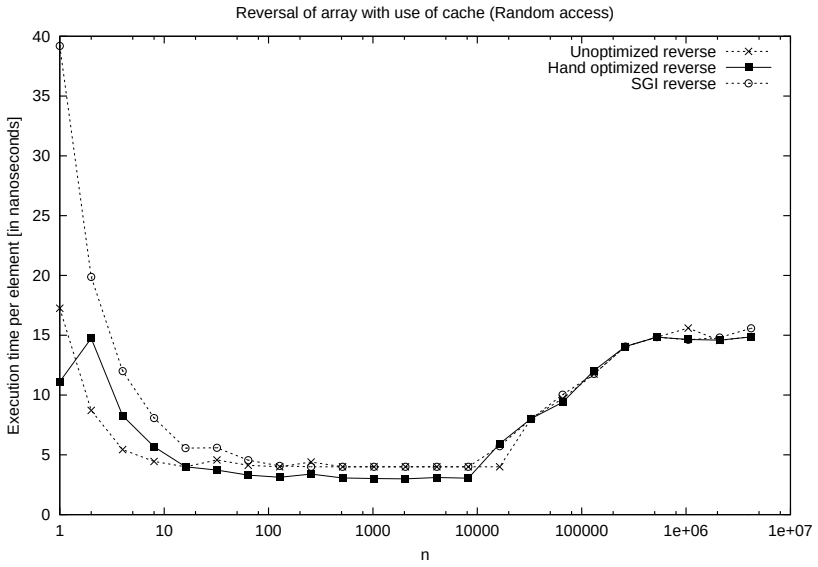

Figure 1. Benchmark for reverse with random access iterators and use of cache.

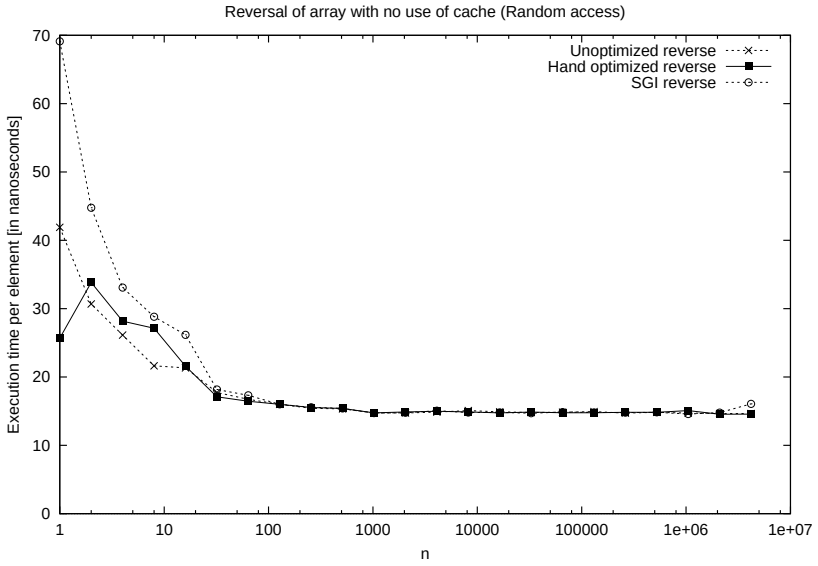

Figure 2. Benchmark for reverse with random access iterators and no use of cache.

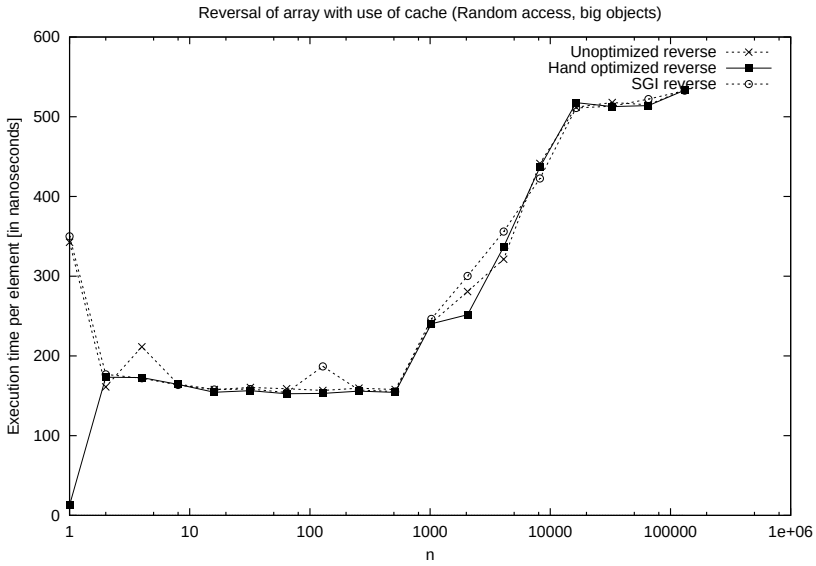

Figure 3. Benchmark for reverse with random access iterators, big objects and use of cache.

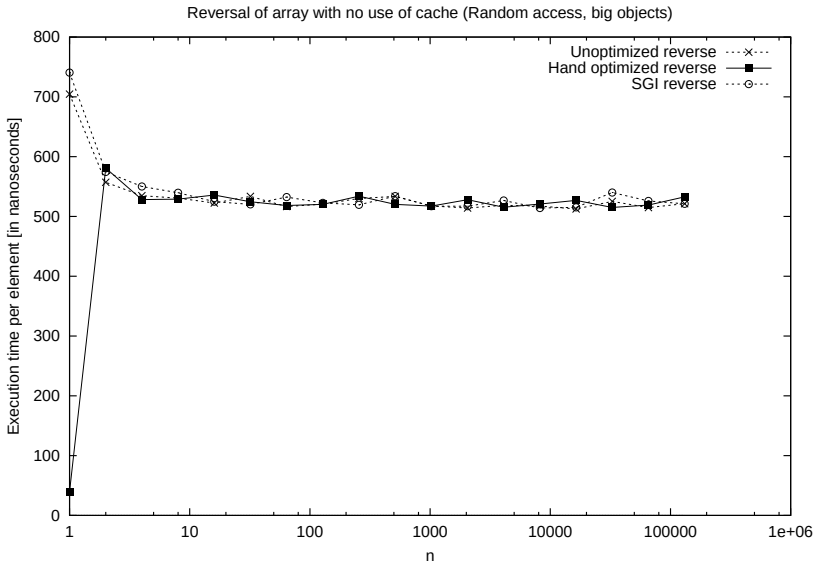

Figure 4. Benchmark for reverse with random access iterators, big objects and no use of cache.

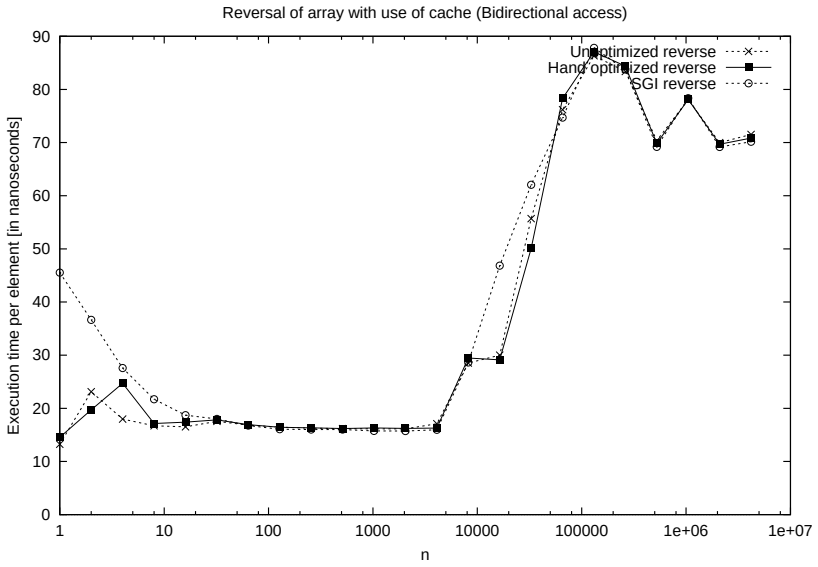

Figure 5. Benchmark for reverse with bidirectional iterators and use of cache.

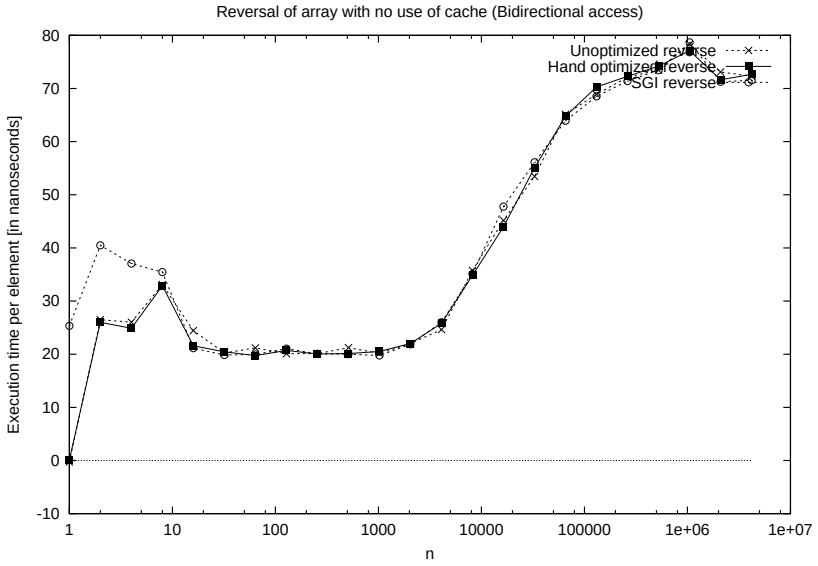

Figure 6. Benchmark for reverse with bidirectional iterators and no use of cache.

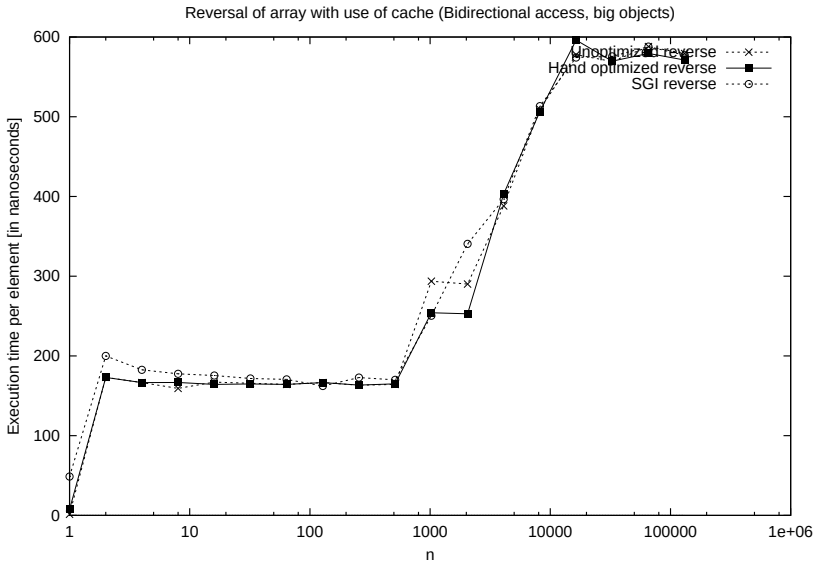

Figure 7. Benchmark for reverse with bidirectional iterators, big objects and use of cache.

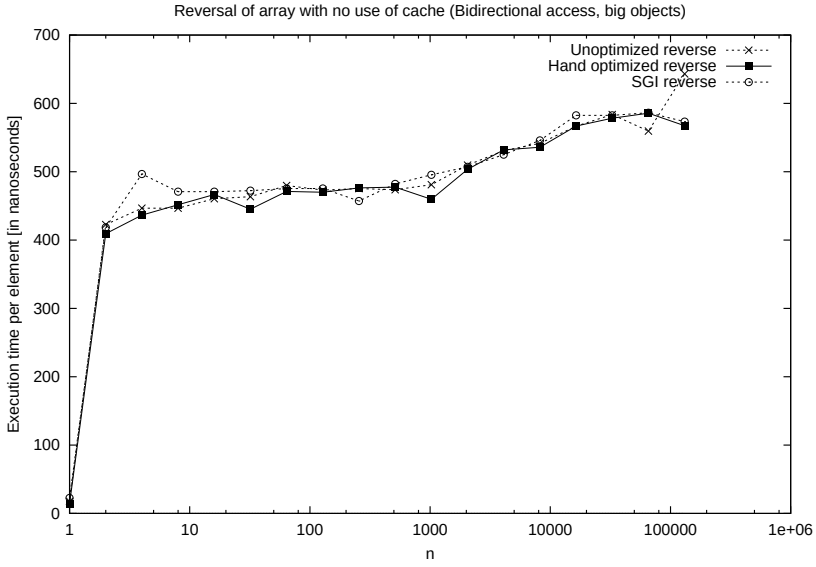

Figure 8. Benchmark for reverse with bidirectional iterators, big objects and no use of cache.

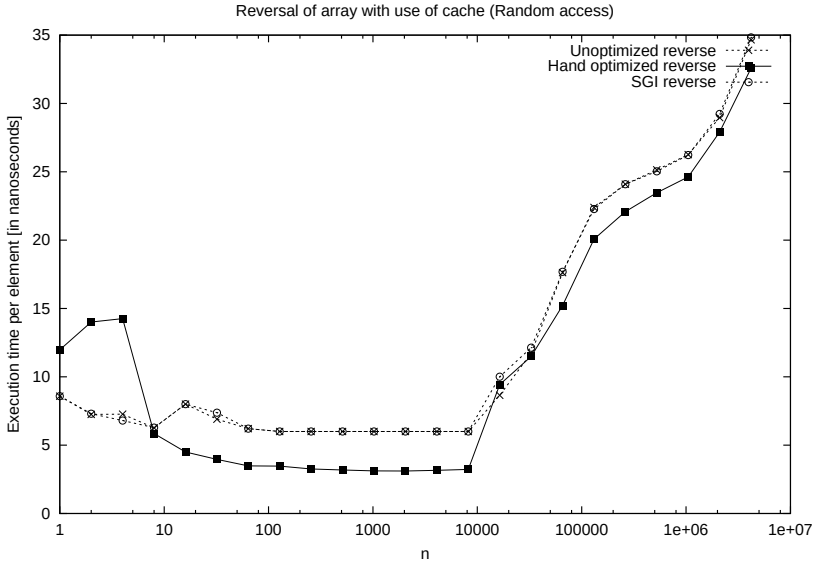

Figure 9. Benchmark for reverse_copy with random access iterators and use of cache.

2. reverse_copy

The reverse_copy takes two bidirectional iterators and an output iterator as arguments. The elements between the two iterators are written to the output sequence in reverse order.

2.1 Algorithm

Some of the same optimizations that where used in reverse, can also be used here. I have split the algorithm in two, one for random access iterators, and one for bidirectional iterators. The random access method I have loop unrolled.

2.2 Benchmarks

The same benchmarks was performed for this algorithm, as for reverse in Sec. 1.2. The unoptimized version is the same as used in [SGI 2001], except that it is inlined.

As suspected, the random access iterator benchmarks gave a performance boost. This is seen in (Figure 9, Figure 10, Figure 11, Figure 12).

Since no optimisation, except the inlining, is used in the bidirectional version. No major difference is expected between the two implementations. Since these benchmarks are not really that interesting, I have only included one benchmark (Figure 13).

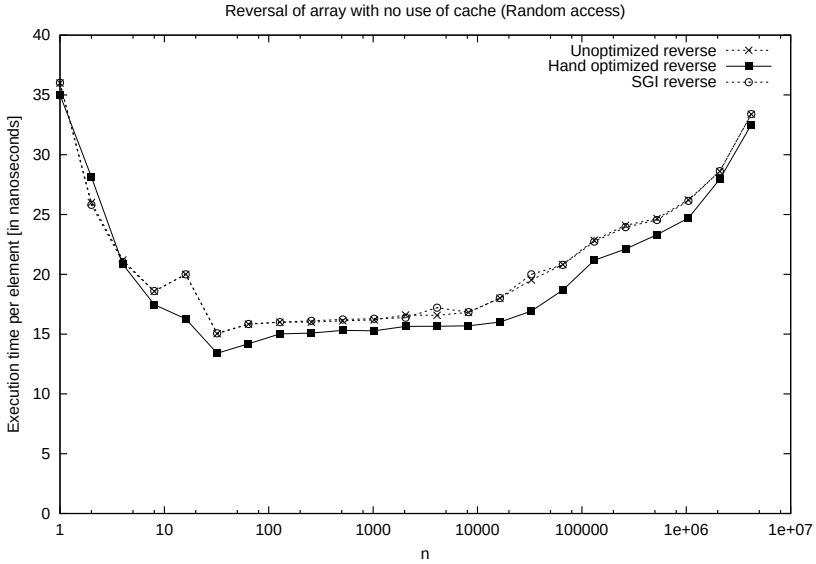

Figure 10. Benchmark for reverse_copy with random access iterators and no use of cache.

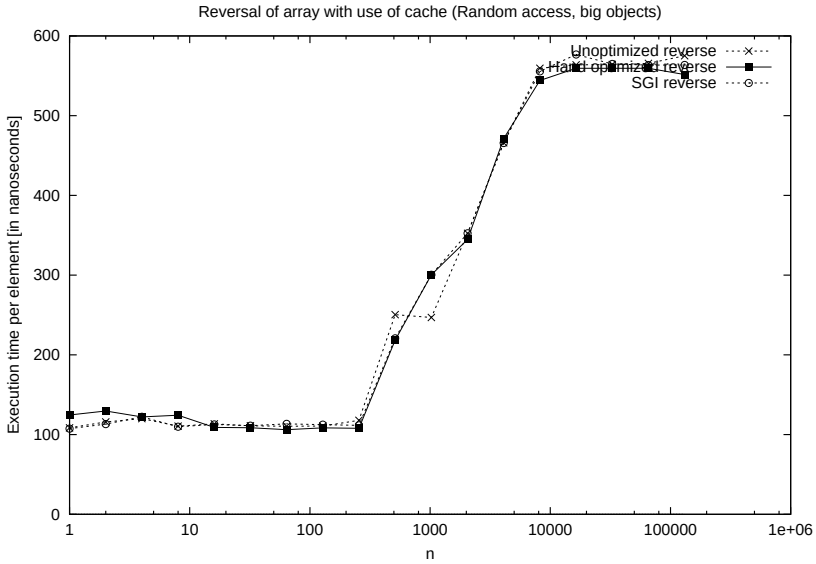

Figure 11. Benchmark for reverse_copy with random access iterators, big objects and use of cache.

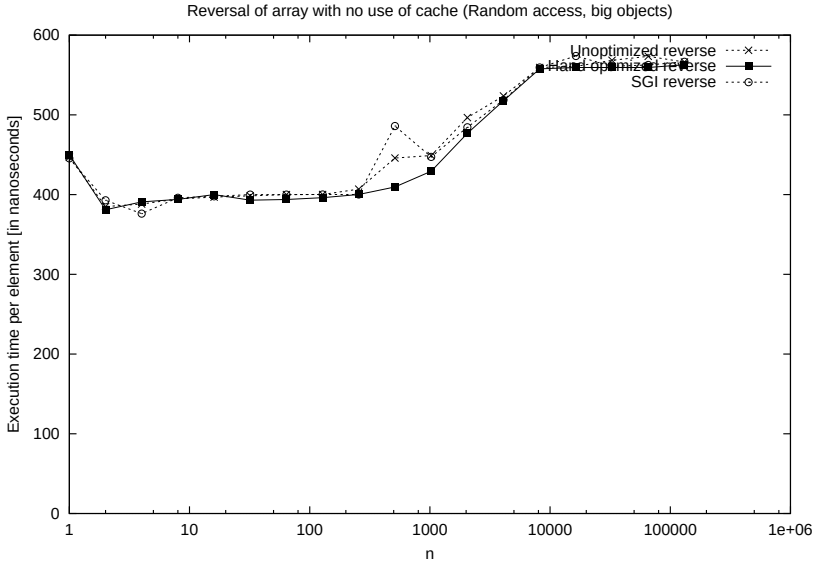

Figure 12. Benchmark for reverse_copy with random access iterators, big objects and no use of cache.

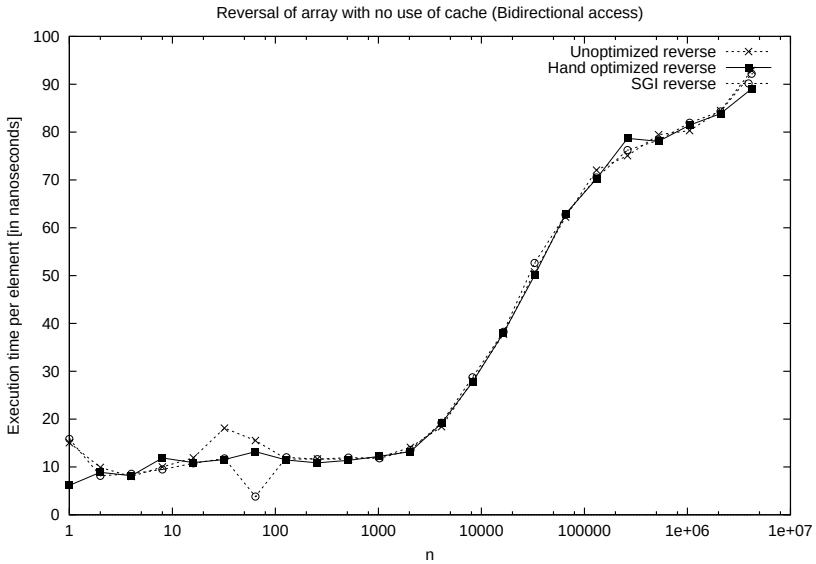

Figure 13. Benchmark for reverse_copy with bidirectional iterators and no use of cache.

3. Forward reverse

The STL usually only supports reverse for bidirectional iterators, and not for forward iterators. The reason for this is that it is expensive to reverse a sequence that only supports forward access and in many cases you have made a bad choice of containers, if you are forced to reverse a container that only supports forward access. However, it is possible to reverse a sequence that only supports forward access, although it is quite a challenge to do it effectively.

The slist class, which is the only sequence in [SGI 2001] that only supports forward access, supports a member function reverse, which reverses the list. This function only edits the internal pointers, and does not copy any of the objects. This function would be better to use in almost any cases. A specialized forward reverse, could be useful in other algorithms which does not have access to the slist object. It could also be useful to reverse shorter segments of list.

Two general algorithms are proposed for this problem:

- A linear time algorithm, which also uses a linear amount of memory.
- A $O(n \log n)$ time algorithm, which works in place and only uses a constant amount of extra memory.

3.1 Linear algorithm

The linear time algorithm copies all of the elements to a temporary buffer, and then copies the elements back again in reverse order. This can however be a problem, if there is no more memory left to use. I will however assume that it is possible to allocate enough memory to execute the algorithms.

3.1.1 Recursive

The recursive function simply takes a copy of the current object, and then calls the function recursively. After the recursive call, it copies the object back to the sequence. In this way the last object will be the first object to be copied back.

In order to optimize this function, the recursive calls can be unrolled.

3.1.2 Non-recursive

The non-recursive function first calculates the size of the sequence, then copies all of the elements to a temporary buffer of that size. The elements are then copied back in reverse order using copy_reverse.

3.2 $O(n \log n)$ Algorithm

The in place algorithm, only uses a constant amount of memory, but the time complexity is worse.

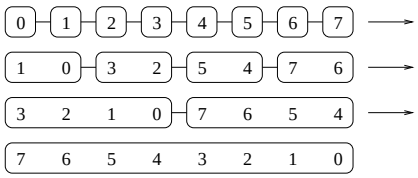

Figure 14. Reversal of array with only use of forward iterators. Simple case where length is a power of two.

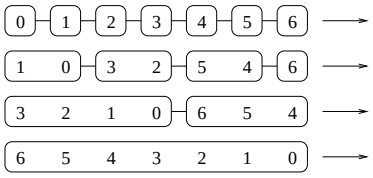

Figure 15. Reversal of array with only use of forward iterators. More complicated case where length is not a power of two.

3.2.1 The basic algorithm

The basic idea behind the algorithm, is to reverse small sequences, and then swap these sequences until the whole array has been reversed. A small example of this can be seen in Figure 14.

If the length of the sequence is not a power of two (in some cases there are other restrictions on the length of the sequence), it is not possible to swap the last two subsequences and they need to be rotated. A small example of this can be seen in Figure 15.

3.2.2 Optimisation tricks

When implementing this algorithm the number of swaps should be minimized. The number of calls to rotate, and the length of the sequence to be rotated should also be minimized.

To reduce the number of swaps, a simple trick, is to reverse sequences of three instead of sequences of two, since the same number of swaps is required to reverse a sequence of three as a sequence of two. Sequences of five and seven is also possible.

To reduce the length of the sequence to be rotated, it might be an idea to switch between reversing sequences of two, three, five and maybe even seven. In this way, a sequence of the length ten could be reversed by a five reversal and a two reversal. And there would not be any bit left unreversed. Where if only three reversal was used, it would take two three reversals, and a rotate. This example is illustrated in Figure 16. The problem in this case is however finding a good way to determine which reversals to use.

If the length of the sequence is known in advance, the operations that should be done can be precalculated, and they can then be done in the

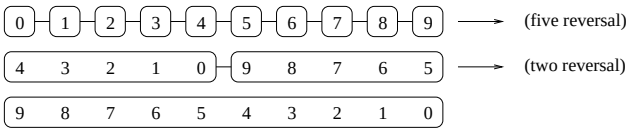

Figure 16. Reversal of array with only use of forward iterators. Bottom-up with different reversal lengths.

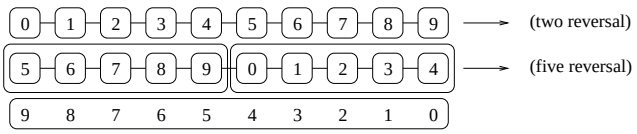

Figure 17. Reversal of array with only use of forward iterators. Top-down with different reversal lengths.

reverse order. This would allow a top-down recursive function. A small example of this can be seen in Figure 17.

3.3 Implementation

I have implemented three different algorithms for this problem.

- The unrolled recursive algorithm as described in Sec. 3.1.1.
- The non-recursive algorithm as described in Sec. 3.1.2.
- The $O(n \log n)$ time algorithm as described in Sec. 3.2.

The unrolled recursive function has been unrolled five levels, but still all data is saved on the stack, which gives the need of at least as much stack space, as the size of the sequence. It is not likely that this amount of stack space is available.

The non-recursive function is implemented straight forward, by first counting the elements, and then allocating temporary space for the elements. This still depends on available free memory, but since it is heap memory and not stack memory, it should be possible to allocate more of it. But still, at some point even the heap memory will be used, and then this is not the best algorithm.

I have implemented the $O(n \log n)$ algorithm as a bottom up algorithm. The basics are as follows: First the sequence is run through and pairs of three elements is reverse. In the end there is either zero, one or two elements left, if there are two elements left these two elements are swapped. During this first run the elements are counted and this count is used in the following iterations. Each of the following iterations is composed of three steps:

- First the subsequences that can be reversed easily is reversed by three reversal, much like in the first run except for the fact that the subsequences that are swapped, are longer than one element.
- Then the remaining part is considered, there is either zero, one or two whole subsequences left. If there is two subsequences left, these two

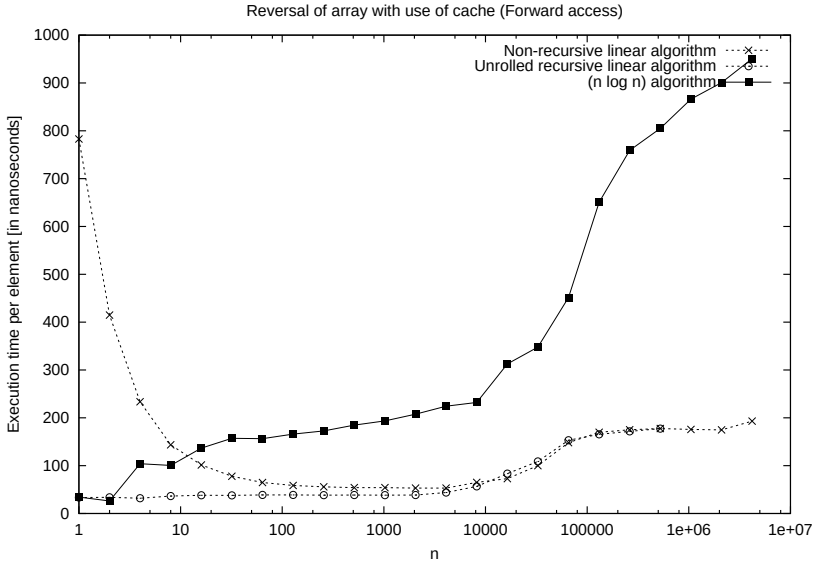

Figure 18. Benchmark for reverse with forward access iterators and use of cache.

subsequences are swapped.

– Lastly the remaining part is rotated with the remainder from the previous iterations.

After $\log_3 n$ iterations the sequence is completely reversed.

3.4 Benchmarks

I have made four benchmarks. One which which runs repeatedly on the same sequence (Figure 18), one which runs on a new sequence each time (Figure 19) and the same two benchmarks for big objects (Figure 20 and Figure 21).

The recursive algorithm seems to be the one that does the best job on all of the four benchmarks. There is however a major problem, when the sequences get too long. The problem is that the algorithm runs out of stack space and fails. This problem makes this algorithm a bad choice.

The non-recursive algorithm is very slow on small sequences, this is because it takes some time to allocate the desired space. This problem can however be fixed, by calling a different algorithm if the length of the sequence is short. A worse problem is however that the algorithm can fail to allocate the desired amount of memory. This is not visible on the benchmarks, but at some point it will happen.

The $O(n \log n)$ algorithm has none of the memory problems that the other algorithms have. The problem with this algorithm is however that it is slower than the other algorithms. Since it is a bottom-up algorithm, it accesses the memory very inefficiently and is therefore slow on large sequences.

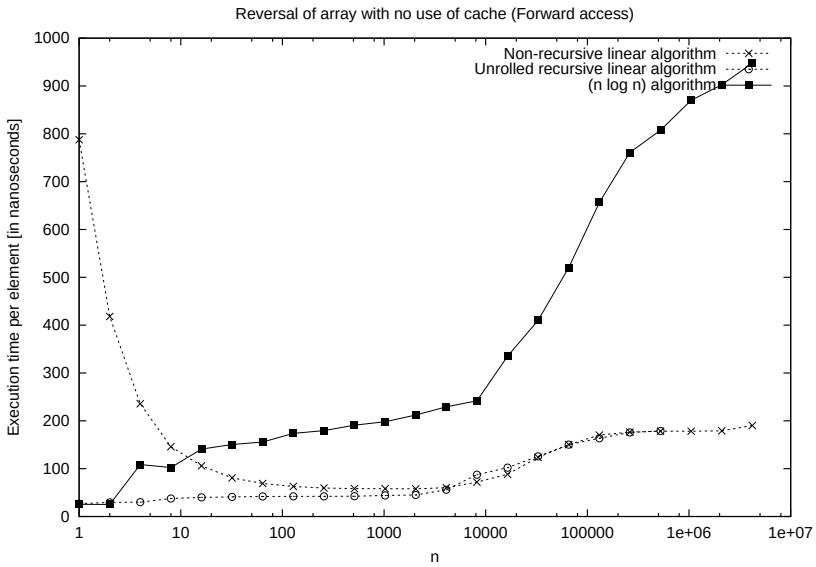

Figure 19. Benchmark for reverse with forward access iterators and no use of cache.

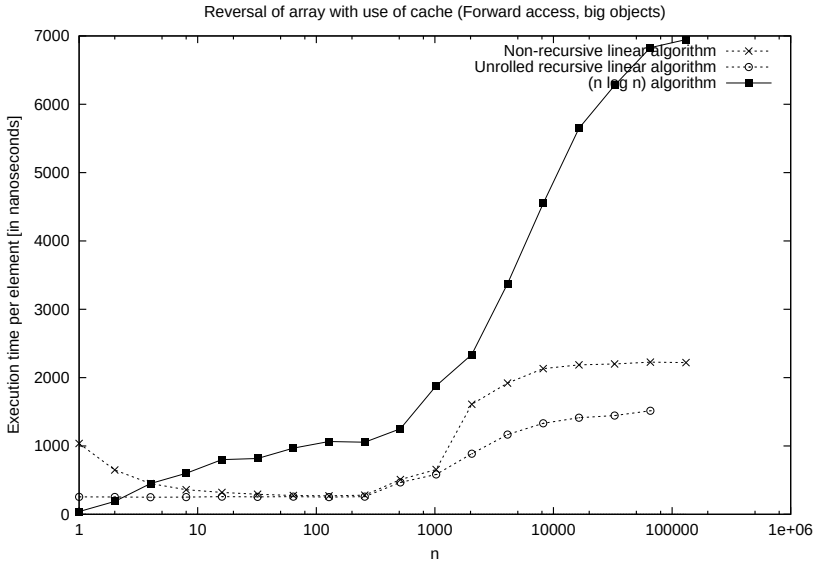

Figure 20. Benchmark for reverse with forward access iterators, big objects and use of cache.

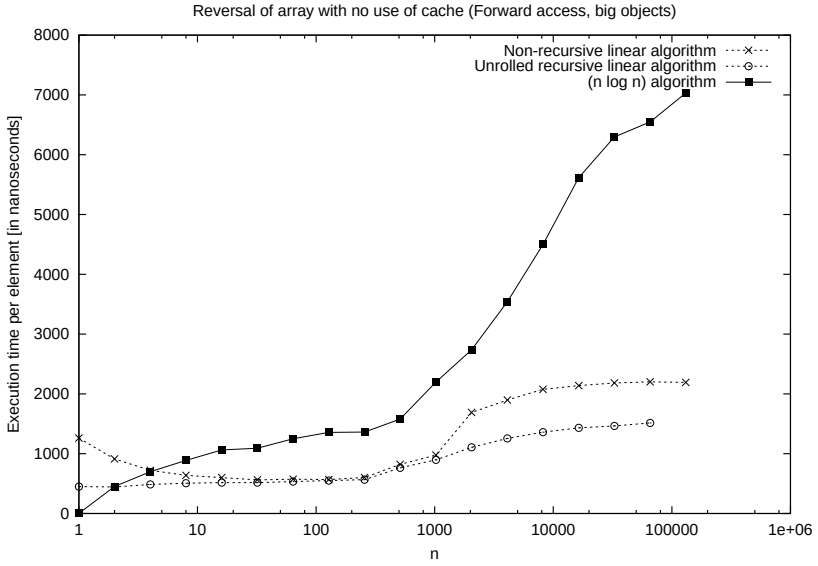

Figure 21. Benchmark for reverse with forward access iterators, big objects and no use of cache.

3.4.1 Conclusion

All of the implemented algorithms has some kind of drawback, which leads me to believe that perhaps none of the implemented algorithms is the right choice for the CPH STL [CPHSTL 2001]. The best algorithm might be a algorithm that uses the $O(n \log n)$ for short sequences and the non-recursive for longer sequences, except when the non-recursive fails to allocate memory, where the $O(n \log n)$ algorithm should be used again. The $O(n \log n)$ algorithm in it self could also be optimized to be top-down instead of bottom-up. And some of the other optimisation tricks could also be implemented.

Implementing this kind of algorithm is a project in it self, which is why I will not do that in this project.

4. next_permutation and prev_permutation

next_permutation takes two bidirectional iterators as arguments, and permutes the sequence between the two iterators. So that it is the next permutation in lexicographical order. E.g. the sequence [a,b,c] will be converted to the sequence [a,c,b]. prev_permutation does the opposite. There is also a version of the two algorithms that takes a comparison object as an argument.

4.1 The algorithm

For this there is only one known effective algorithm. It is an algorithm proposed by Fischer and Krause back in 1812. More can be read about this

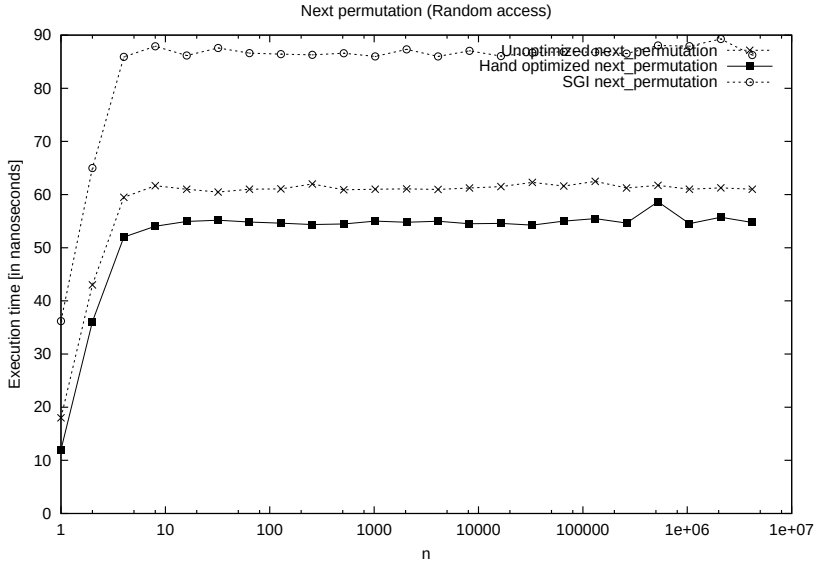

Figure 22. Benchmark for next_permutation with random access iterators.

algorithm in [Sedgewick 1977a], [Sedgewick 1977b] and [Johnson 1998]. I will not explain how or why this algorithm works, but it contains a call to reverse, for sequences, which most of the time are very small. The algorithm is not effected much by the length of the sequence, since most of the time only the least significant elements are changed.

4.2 Optimizations

The most straight-forward optimisation is to use my own new optimized reverse, and to make the function inlined. Another is splitting up the algorithm in one for random access iterators, and one for bidirectional iterators. The random access version can furthermore be optimized.

4.3 Benchmarks

I have chosen to run the program a large number of times and calculated how long time it took to run the algorithm at a specific length, as opposed to calculating the time per element. This I have done, because the algorithm is not affected much by the length of the sequence. Because of this choice, I can only show benchmarks, which is run repeatedly on the same sequence, causing the elements to be in the cache.

Since there is no significant difference between the next_permutation and the prev_permutation implementations, only benchmark results for next_permutation are showed. Benchmarks for the version which takes a comparison object is not showed either, since it only gives a small penalty.

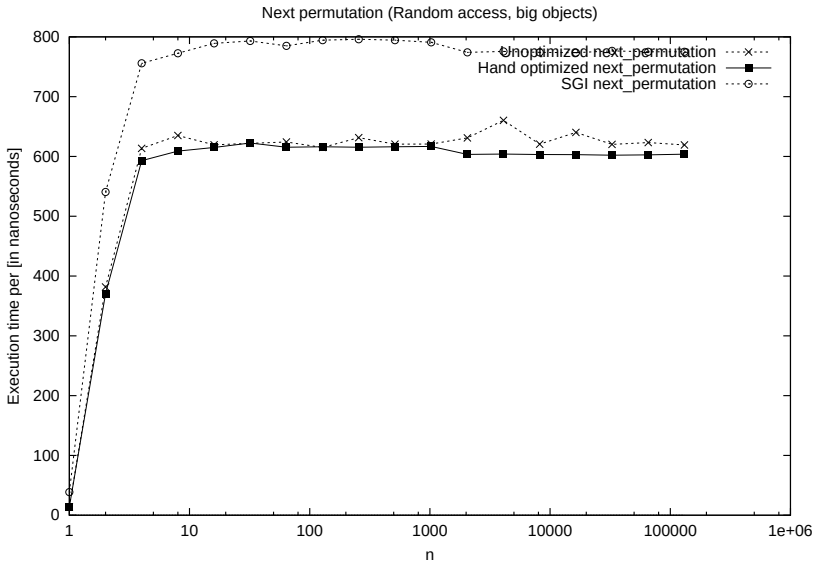

Figure 23. Benchmark for next_permutation with random access iterators and big objects.

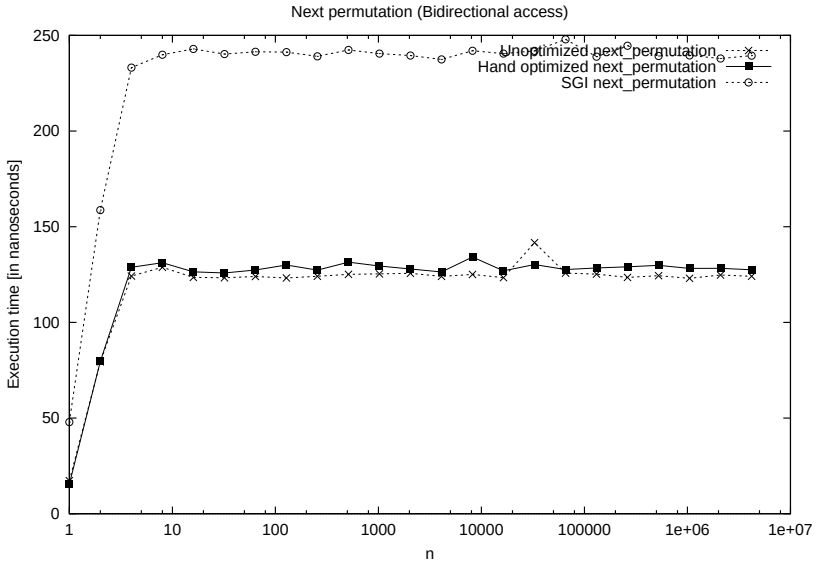

Figure 24. Benchmark for next_permutation with bidirectional iterators.

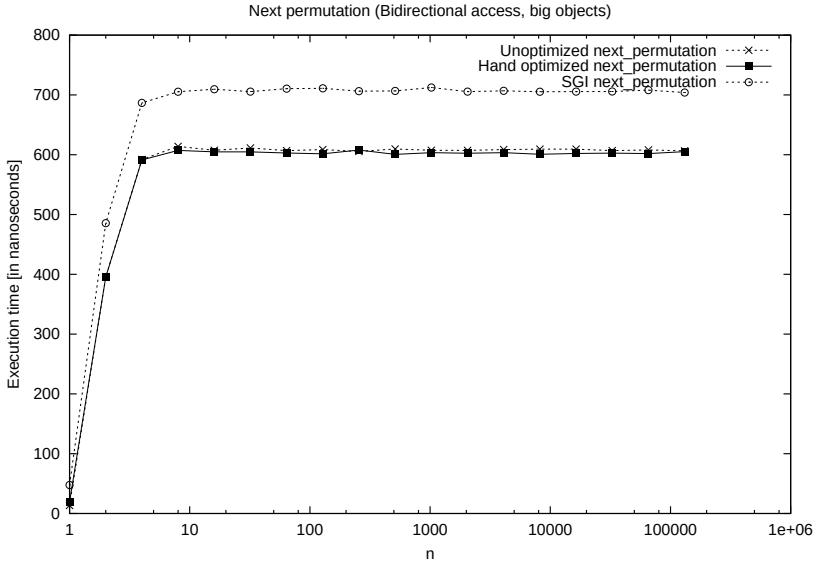

Figure 25. Benchmark for next_permutation with bidirectional iterators and big objects.

The unoptimized version is exactly the same as the SGI STL version [SGI 2001], but with the use of my optimized reverse and inlining.

For random access iterators a big performance boost can be seen on Figure 22 and on Figure 23.

For bidirectional iterators a performance boost can be seen on Figure 24 and on Figure 25. But a strange thing has happened, the unoptimized version are faster than the hand optimized version. The reason for this, is that the two version i practically the same, but the unoptimized version is not split in a random access version and a bidirectional version. Which apparently helps a bit in the inlining, and gives a slightly smaller overhead.

5. Conclusion

The main purpose of this paper was to implement the next_permutation and the prev_permutation algorithms for the CPH STL [CPHSTL 2001]. Along the way came other projects like reverse, reverse_copy and forward reverse.

The main project was a success and the permutation algorithms are faster in [CPHSTL 2001] than in [SGI 2001]. The other projects where also a success, except the forward reverse algorithm which I do not think is mature enough to be included in the Copenhagen STL.

5.1 Future work

I do not think that there is much future work in any of the algorithms that I have implemented, except for forward reverse. Much work can be done for

forward reverse, but there has not been found any real use for this function see discussion in Sec. 3. I do not think that too much work should be put into forward reverse before a real need for this function is established.

References

Bojesen, Jesper. 2000. Managing Memory Hierarchies. Master's thesis, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark.

CPHSTL, DIKU Performance Engineering Laboratory. 2001. The Copenhagen STL http://www.cphstl.dk/. Internet document.

Johnson, Jeffrey A. 1998. Revisiting Lexicographical Permutation Methods. Tech. report, Brigham Young University, Hawaii.

Mortensen, Sofus. 2001. Refining the pure-C cost model. Master's thesis, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark.

Sedgewick, Robert. 1977a. Permutation Generation Methods. *9*, 2 (June), 137–164.

Sedgewick, Robert. 1977b. Corrigenda: "Permutation Generation Methods". *9*, 4 (Dec.), 314–314.

SGI, Silicon Graphics inc. 2001. Standard Template Library Programmer's Guide http://www.sgi.com/tech/stl/index.html. Internet document.