FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Assignment 3: Programopførsel

CompSys 2017
Department of Computer Science
University of Copenhagen

Peter, Tobias & Xueying

29. October 2017

1 Compiling and running the code

1. Unzip `src.zip` in the same directory as `architecture-tools`.
2. The programs may be assembled and simulated via a UNIX-like command line in the following way:
   (a) Navigate to the directory containing `architecture-tools` and `src`.
   (b) Assemble a program by running this command:
   `./architecture-tools/asm ./src/file.x64 ./src/file.out`
   (c) Simulate a program by running this command:
   `./architecture-tools/sim ./src/file.out`
   (d) Model the performance of a program by adding the `-m`-flag:
   `./architecture-tools/sim ./src/file.out -m`
   (See all flag options by leaving out the 'm').
3. There are no tests available.

2 Measuring the matrix multiplication program

We have measured the number of cycles when the size of matrix changes from 10, 15, 20, 25, 30, 35, 40, 45 to 50 and here is the graph of the gathered data:

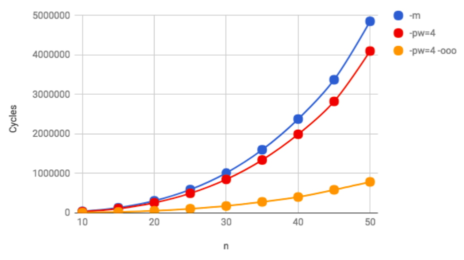

Figure 1: Total cycles among different machines

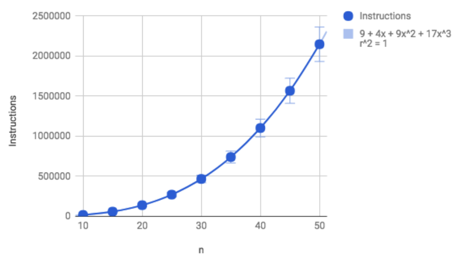

Figure 2: Number of instructions based on n

As illustrated above, the 4-ways superscalar pipeline with out-of-order execution has a much better performance especially when the matrix size goes up.

We have also compared the total instructions and the expected run-time $O(n^3)$ which are identical to each other. As seen on figure 2, the graph's $R^2 = 1$ which means that the function is the shape of the third degree polynomial:

$$9 + 4x + 9x^2 + 17x^3$$

Since the polynomial's highest degree is 3 (The $17x^3$ has the highest growth rate) then it means that the algorithm must run in $O(n^3)$.

From the gathered data, we can get the constant K, this has been done by isolating K in the equation: number of cycles $= K \cdot n^3$ for all $n = 10, 15...50$, and then computing the mean of these constants (K).

Models	-m	-pw=4	-pw=4 -ooo
K	38.2082	32.1105	6.6731

Table 1: The average K value from data of the three different machines

3 Measuring the sorting programs

We have measured the number of cycles when the size of array changes from 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000 to 10000 and here is the graph of the gathered data:

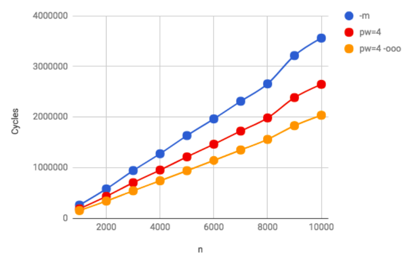

Figure 3: Mergesort - Total cycles among different machines

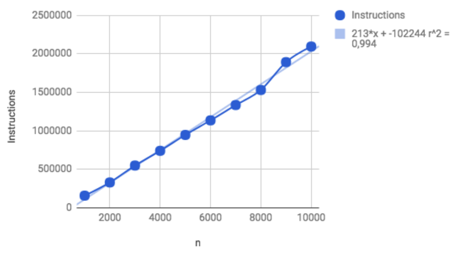

Figure 4: Mergesort - Number of instructions based on n

Figure 4 shows a graph over the number of instructions carried out by the merge sort algorithm. Just by looking at the graph it is not possible to distinguish whether the code runs in $O(n)$ or $O(n\log(n))$, sine the graph appears like a linear function. Therefore we must also take a look at the code. By looking at the code one can see that it follows the divide and conquer pattern of other merge sort algorithms. Together with the graph we can now be sure that the code actually runs in $O(n\log(n))$ and not in $O(n)$ time which the graph could indicate.

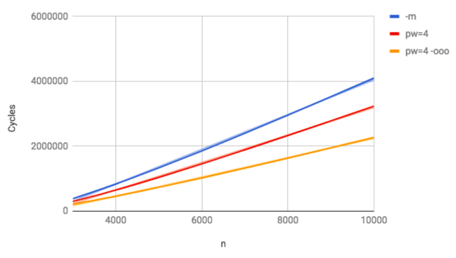

Figure 5: Heapsort - Total cycles among different machines

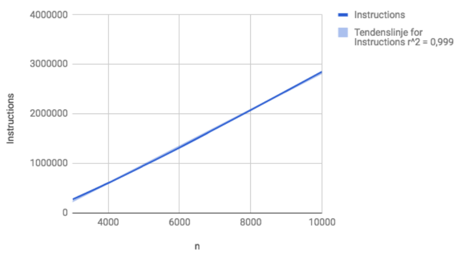

Figure 6: Heapsort - Number of instructions based on n

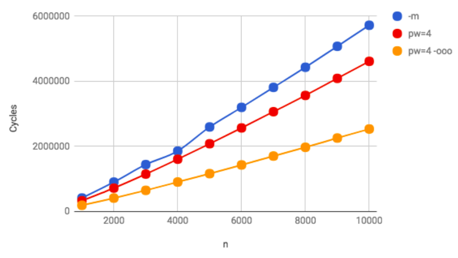

Figure 7: Heapsort2 - Total cycles among different machines

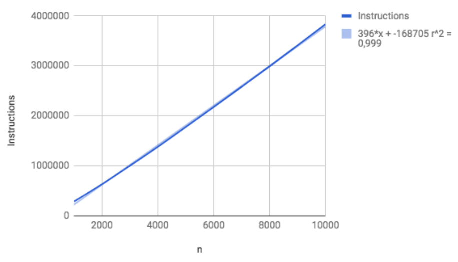

Figure 8: Heapsort2 - Number of instructions based on n

As in the case of the mergesort-program, we cannot conclude that the heapsort-program runs $O(n\log(n))$ based on our graphs alone. Looking at the code one may notice that the `heapsort` function first calls $sift_{down}$ roughly $start = (count - 1) >> 1 \approx \frac{n}{2}$ times, where $sift_{down}$ runs for $\log n$ as it "builds" a heap (tree) from the array-elements. The `heapsort` function then calls $sift_{down}$ $end = (count - 1) = n$ times. We then have a total time complexity of $\frac{n}{2}\log n + n\log n O(n\log n)$.
Comparing heapsort and heapsort-2, one will notice that heapsort performs better than heapsort-2 on average.

Models	heapsort	heapsort-2	mergesort
-m	39.0901	41.9433	26.4261
-pw=4	30.7249	33.8735	19.6245
-pw=4 -ooo	21.5982	18.9864	15.3088

Table 2: The avg. K based on the data from the three different machines when using the different sorting programs

4 A quicker matrix multiplication

We decided to optimize `matrixmult-big` by changing the order of the loops from an *ijk*-ordering to an *ikj*-ordering, thus achieving better locality - a necessary improvement considering that our largest test-matrix does not fit in the L1-cache.

```
void multiply(int dim, int mat1[dim][dim], int mat2[dim][dim], int out[dim][dim])
{
    int i, j, k;

    for (i = 0; i < dim; i++)
    {
        for (k = 0; k < dim; k++)
        {
            r = A[i][k];
```

```
                for (j = 0; j < dim; j++)
                {
                    out[i][j] += r * mat2[k][j];
                }
            }
        }
    }
```

In practice, this ordering allows the machine to access both the second matrix/array and the output-matrix/array *row-wise*, rather than column-wise.

4.1 Performance

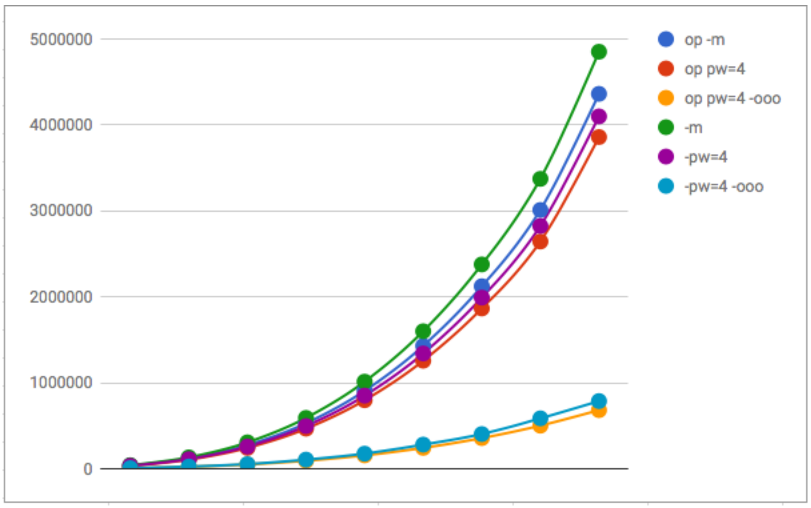

Figure 9: Measured cycles of original `matrixmult-big` and optimized `matrixmult-big` ("op"-series).

4.2 Constants

Models	-m	-pw=4	-pw=4 -ooo
K	34.2424	30.1407	5.9219

Table 3: Approximated, average K-values for `matrixmult-big-op` on the three different machines.

4.3 Conclusion

Our optimized version of matrixmult-big avoids jumping long distances through the matrix/array, which would result in more cache misses. This is evident from our modelling-results (see graph and constant-table) which show that our optimized program waits for fewer cycles (and performs slightly fewer instructions, although this is not included in the graph). Further optimization could have been achieved through loop-unrolling the inner j-loop, making changes to address computations and performing multiplication through more efficient operations.

5 A quicker sorting program

In this section we where asked to either create a new program or optimizing one of the sorting programs. Our approach to create a quicker sorting program, was to optimize the first heapsort program. We have earlier seen the performances of the three given programs and mergesort has a shorter run-time, thus heapsort has more possibilities.

Seeing that there are two big while-loops in the heapsort function, to improve one of the loops was our first thought.

Cutting the array into half and loop through these two half part simultaneously was one of our attempts. Ideally, the run-time of the loop will be reduced by half. With the cutting half idea in mind, we tried different approaches and finally landed on the one that making start point counting double so fast down to 0.

5.1 Performance

In comparison with the original heapsort program, the optimized version shows the similar tendency graphically:

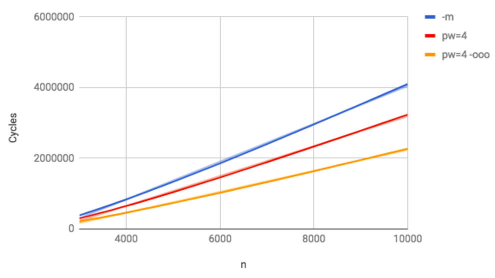

Figure 10: Heapsort - Total cycles among different machines

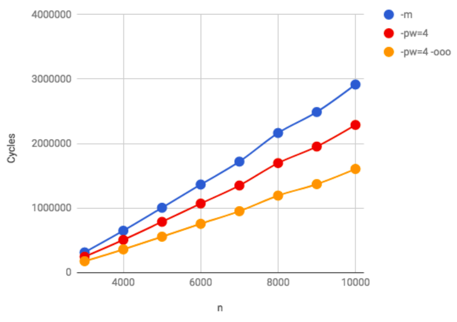

Figure 11: Optimized version of heapsort - Total cycles among different machines

Compared to the old heapsort the new heapsort runs fairly faster as the graph also indicates.

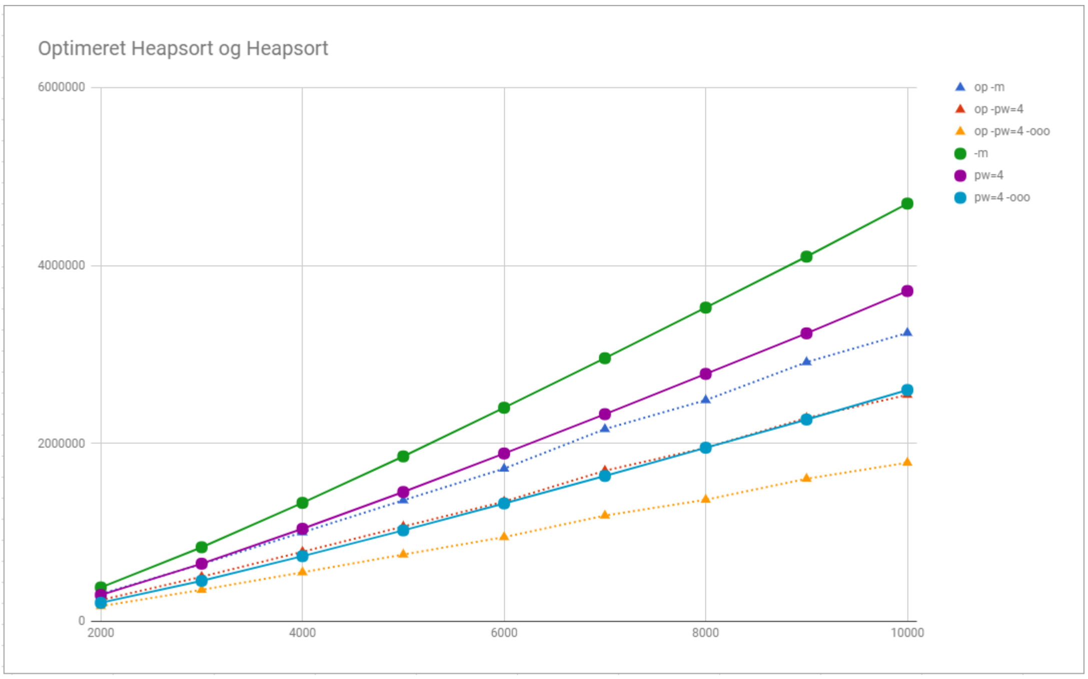

Figure 12: Measured cycles of original `heapsort` (The circles) and the optimized `heapsort` ("op"-series/triangles).

5.2 Constants K

Models	heapsort	heapsort-op
-m	39.0901	28.4952
-pw=4	30.7249	22.3310
-pw=4 -ooo	21.5982	15.7343

Table 4: The avg. K from data of the three different machines when using two sorting programs

5.3 Conclusion

The optimized program has both fewer instructions and cycles, which means that it runs faster, despite having a greater average latency. It also sports a lower K value, which indicates a quicker performance. To further optimize the program, one could consider scaling the pointer once and for all.

Our optimized heap-sort is still not better than the handed-out merge-sort. The reason behind this is that the merge sort algorithm had far better memory locality compared to our optimized heap-sort, which is often the case. Implementing a quicksort algorithm instead of optimizing, would might have been a better solution.