PX425 Assignment 2 (`burgers.c`)

`phuwcs` (1833194)
Department of Physics, University of Warwick
(Dated: November 12, 2021)

Application of techniques taught in PX425 lectures pertaining to optimisation of serial code. This program implements the 2D Burgers' equation, used to model the evolution of a velocity field in a fluid.

CONTENTS

I. FOREWORD

For compilation, the overall fastest result is using `clang -std=c99 -lpng -ffast-math -mavx -mavx2 -march=native -O3 burgers.c makePNG.c mt19937ar.c`. The same flags produced the fastest performance for `gcc`, although it was slower than `clang` (III).

The versions of compilers used are shown in table I. Minor changes were made to `makePNG.c` in order to make it compile with `-Werror` enabled.

A `Makefile` is included which has the used compiler flags, and `make CC=clang` (for example) can be used to configure the C compiler.

Compiler	Version
`gcc`	10.2.0
`clang`	11.0.1
`icc`	19.0.5.281

Table I. Versions of used compilers.

II. CHANGES MADE

This list details each change made to the program in order. Timings are available on page 6. Intermediate versions of the program are stored as a private GitHub repository for benchmarking.

1. Usages of the `pow` function (from `math.h`) were removed and replaced with squaring the variable. For example, `pow(dx, 2.0)` became dx * dx. This was because all instances of `pow` were for squaring variables which is trivially replaceable with simple multiplication. Furthermore, since `dx` and `dy` do not change over the course of the program these values can be safely precalculated immediately after `dx` and `dy` are set. This also removed the requirement to link the math library.
This change had a large impact on the performance at `-O0`, but no effect on the optimised program. This is because the optimiser already factorised out instances of `pow` and stored a single result.

2. When `u_new` is moved into `u`, swap pointers of the two arrays rather than copying the values one-by-one. That is to say:

```
double** tmp = u_new;
u_new = u;
u = tmp;
```

This change was made because simply put there is no point doing it any other way. Copying values one-by-one is an $O(n^2)$ algorithm. Using `memcpy` to copy each row of the array would be $O(n)$. Swapping pointers is $O(1)$. Since this swap is also the last part of each iteration, there is no risk of reading incorrect data or writing to the wrong place when this swap is made.
This change resulted in good improvements at all optimisation levels.

3. Factorise out division by dx^2 and dy^2 in the calculation of the Laplacian and the grad operator. This reduces the number of division operations made in each equation.
Small improvement at `-O0`, not with optimisation passes for the same reasons as 1.

4. Replace all division with multiplication by a reciprocal. This involved replacing some instances of `/ 2.0` with `* 0.5`, and also precalculating values for $\frac{1}{\mathrm{d}x}$, $\frac{1}{\mathrm{d}y}$, $\frac{1}{\mathrm{d}x^2}$, and $\frac{1}{\mathrm{d}y^2}$. This change replaced an expensive operation (division) with a less expensive one (multiplication) while also avoiding unnecessary work.
   Interestingly this had almost no effect despite being somewhat pervasive across the program, with only a small improvement at `-O0`.

5. Swap the order of the loop counters in the main loop so that iy is the fastest varying counter. This is because of the way the array is indexed. Since the second (inner) index is the y index, columns are located in contiguous blocks in memory, while there is no guarantee that each row is spatially close. This change reduces the number of cache misses that occur in the main loop.
   This change was effective at all optimisation levels. Interestingly, `-O3` enables the `-floop-interchange` flag which should be able to automatically make this change. Branching in the loop likely made the compiler unable to perform this optimisation automatically.

6. Precalculate $\frac{1}{2\mathrm{d}x}$ and $\frac{1}{2\mathrm{d}y}$. These were the last terms derived from dx and dy in the equations. This change was made for the same reason as the previous precalculation changes: to reduce the amount of repeated work when the result is already known.
   As with previous precalculation and factorisation efforts this had a minimal effect on the code's speed.

7. Remove branching in the main loop. The boundary conditions on the equation can be divided into three regimes: the four corner elements of the grid, the elements along the edges, and the interior elements. Each of these regimes had a different method applied to remove branching.
   (a) For the corners, the four cases were handled specifically.
   (b) For the edges two loops were used, one handling the left and right edge as iy varies, and the other handling the top and bottom edge as `ix` varies.
   (c) For the interior, the original approach was used.

   This meant in each case no branching was required, as only one pair of conditions from the two `if` statement chains in the original code were true at any one time. Hence in the remaining three `for` loops, branching conditions could be removed and replaced with the only pairs of contributions that could be met in that case. Furthermore, instances of `ix` and `iy` could be replaced with `0`, and similar for other things that are now constants.
   Removing branching sped up the code at everything except `-O2`, where performance stayed the same as before.

8. Applying the `const` qualifier to variables that do not change during the program. The intention was that with additional information about variables not changing during the program they could be inlined by the compiler. Inspecting the emitted assembly code shows the compiler has decided to put these values into XMM registers instead.
   No tangible effect on performance. This change was kept because it makes it clear that these variables do not change.

9. Precalculation of loop terms. The loop without branching contained many references to, for example, `u[0][0]`. While on higher optimisation levels this was already stored and re-used, this was not the case on `O0`. Each instance of this was extracted to its own variable to prevent the program dereferencing the same pointers multiple times.
   As with the earlier precalculation attempts this did not produce a meaningful impact on the program's execution time, but was kept because it made the equations easier to read.

10. Disable creation of snapshot `.png` files unless the `NDEBUG` macro is defined (for example by `-D NDEBUG` during compilation). Analysis of a flame graph showed that a small amount of time was used calling this function so it was disabled at higher optimisation levels to save time.
    No significant change in execution time. Writing the `.png` file is overshadowed by the algorithm being O(n^2).

11. Enabling the `-ffast-math` compiler flag. This compiler flag in turn enables several others[1]. The effects of this flag are shown on page 5. Enabling this flag allowed the compiler to perform more performant optimisations on the equations involved. The loss of precision in the low-precision bits is not relevant to this program since its output is only to five significant figures.
    This change lead to a decrease in execution time when the code was optimised, since this flag does nothing at `-O0`. It was an effective change since the final output of the program is still correct.

12. Enabling the `-mavx2` compiler flag. This flag allows the compiler to emit `avx2` SIMD instructions. These instructions use 256-bit registers in the CPU capable of transforming 4 double precision floating-point numbers at once, providing a significant increase in speed.
    As with above, this change only had an effect when optimisation was enabled. Leveraging `avx2` in-

structions provided good increase in performance at `-O3` but not much below that.

13. Enabling the `-mavx` compiler flag. This was enabled in case the compiler preferred to use 128-bit registers with the same function as above in places over the 256-bit `avx2` registers.
    This flag had a slight improvement in the `-O1` case.

14. Enabling the `-march=native` compiler flag. This specifies that the compiler should generate assembly code targeting the specific CPU architecture running on the machine the program is compiled on. This was enabled as it allows the compiler to perform more specific optimisations and emit more specific instructions, which are better suited to the machine the program runs on.
    Interestingly this made the program slower for `gcc` on `-O3` for unknown reasons.

15. Changing the memory allocation regime to allocate a contiguous block of the total required size. Instead of allocating many nested arrays, allocate one large array and then set regions of the outer array to offsets of the inner one. This approach combines the best of both the rejected 2D array (III) and the nested approach, namely spatial locality and access via only two dereferences.
    This change increased the execution time at `-O0` and `-O1`, potentially because the program was not making fast enough use of the values in the cache (due to a lack of `avx` instructions at this optimisation level?) requiring them to be loaded again.

16. Using preprocessor directives to change the memory management method based on whether the `__OPTIMIZE__` macro is defined or not. This mitigates the performance penalty on `-O0` of 15 while retaining its benefits when the optimiser is running. This was an effective change, keeping the desirable behaviour in most cases.

17. Further factorisation of the equation for the next value of `u` (`u_new[ix][iy] = uxy - dt * uxy * grad + dt * nu * Lapl`). Counting the floating point operations here, there are 4 multiplications and 2 additions/subtractions. Factorising out dt: `u_new[ix][iy] = uxy - dt * (uxy * grad - nu * Lapl)` and there now only 3 multiplications and 2 additions/subtractions.
    For `gcc` change had an improvement on `-O0`. At `-O1` and `-O2` there was no improvement - the compiler should be using `-ffast-math` to apply this automatically. Interestingly there was still a performance gain at `-O3`. Evidently this helped the compiler determine a more optimal way of doing the calculation.

III. REJECTED CHANGES

Some changes were made that did not result in a significant change of execution time, and were discarded. These are listed here.

- Using pre- rather than post-increment operators. It is sometimes said (mostly for older compilers) that the pre-increment operator (`++x`) is more efficient than the post-increment operator (`x++`). This is because compilers retained a copy of the previous value so it could be referenced later. After changing all occurrences of this in the program, no change in execution time was measured. Modern compilers do not create this copy if the value is unused.

- Using a flattened 2D array rather than a nested one. A flat 2D array implementation was trialled as follows:

```
typedef struct {
        double* values;
        int n_rows;
        int n_cols;
} array2d_t;
```

  By flattening the array, rather than having nested arrays, the entire grid of values is spatially close. However, it was found that the overhead in accessing elements was not worth the gains in spatial locality. This overhead comes from the way that values are stored in the grid. To access the element at i, j the program must compute $ix+j$ where x is the number of rows, then dereference that offset. This is two operations followed by a pointer dereference. When using nested arrays, the program only has to perform two dereferences with no addition or multiplication, which is significantly faster.

- Replacing `calloc` calls with `malloc`. `calloc` is slower since it takes time to initialise memory to zero values, however this was not measurable as memory allocation does not take a significant amount of time in this program. This change was rejected because the signature and semantics of `calloc` is preferred by the author. Furthermore, `calloc` may allocate more memory than requested to fit alignment requirements, which can be beneficial.

- Manually applying an unroll-and-jam optimisation to mitigate pipeline stalls. This is handled automatically by the compiler at `-O3`. There was no measureable improvement when this was added, presumably because the loop already has enough instructions to prevent a stall. Examining the emitted assembly (III) showed there are plenty of instructions per loop already.

TABLES AND GRAPHS

Contents of the interior loop, when compiled with `-fverbose-asm -Wall -Wextra -Wpedantic -std=c99 -ffast-math -mavx2 -mavx -march=native`.

```
.LBB2_13:                                    #
Parent Loop BB2_5 Depth=1
        vmovupd ymm1, ymmword ptr [rax + 8*rdi + 8]
        vmovupd ymm2, ymmword ptr [rdx + 8*rdi + 8]
        vmovupd ymm3, ymmword ptr [rsi + 8*rdi]
        vmovupd ymm4, ymmword ptr [rsi + 8*rdi + 8]
        vmovupd ymm5, ymmword ptr [rsi + 8*rdi + 16]
        vbroadcastsd
ymm0, qword ptr [rip + .LCPI2_2] # ymm0 = [-4.0E+0,-4.0E+0,-4.0E+0,-4.0E+0]
        vmovapd ymm6, ymm0
        vfmadd213pd
ymm6, ymm4, ymm1 # ymm6 = (ymm4 * ymm6) + ymm1
        vaddpd  ymm7, ymm2, ymm5
        vaddpd  ymm6, ymm6, ymm7
        vaddpd  ymm6, ymm6, ymm3
        vsubpd  ymm1, ymm1, ymm2
        vaddpd  ymm1, ymm1, ymm5
        vsubpd  ymm1, ymm1, ymm3
        vbroadcastsd
ymm2, qword ptr [rip + .LCPI2_9] # ymm2 = [-5.0E-1,-5.0E-1,-5.0E-1,-5.0E-1]
        vmulpd  ymm3, ymm4, ymm2
        vmulpd  ymm3, ymm3, ymm1
        vbroadcastsd
ymm1, qword ptr [rip + .LCPI2_4] # ymm1 = [5.0E+0,5.0E+0,5.0E+0,5.0E+0]
        vfmadd213pd
ymm6, ymm1, ymm3 # ymm6 = (ymm1 * ymm6) + ymm3
        vbroadcastsd
ymm3, qword ptr [rip + .LCPI2_5] # ymm3 = [1.0E-3,1.0E-3,1.0E-3,1.0E-3]
        vfmadd213pd
ymm6, ymm3, ymm4 # ymm6 = (ymm3 * ymm6) + ymm4
        vmovupd ymmword ptr [rcx + 8*rdi + 8], ymm6
        cmp     rdi, 248
        je      .LBB2_14
        vmovupd ymm4, ymmword ptr [rax + 8*rdi + 40]
        vmovupd ymm5, ymmword ptr [rdx + 8*rdi + 40]
        vmovupd ymm6, ymmword ptr [rsi + 8*rdi + 32]
        vmovupd ymm7, ymmword ptr [rsi + 8*rdi + 40]
        vmovupd ymm8, ymmword ptr [rsi + 8*rdi + 48]
        vfmadd213pd
ymm0, ymm7, ymm4 # ymm0 = (ymm7 * ymm0) + ymm4
        vaddpd  ymm9, ymm8, ymm5
        vaddpd  ymm0, ymm9, ymm0
        vaddpd  ymm0, ymm0, ymm6
        vsubpd  ymm4, ymm4, ymm5
        vaddpd  ymm4, ymm8, ymm4
        vsubpd  ymm4, ymm4, ymm6
        vmulpd  ymm2, ymm7, ymm2
        vmulpd  ymm2, ymm2, ymm4
        vfmadd213pd
ymm1, ymm0, ymm2 # ymm1 = (ymm0 * ymm1) + ymm2
        vfmadd213pd
ymm3, ymm1, ymm7 # ymm3 = (ymm1 * ymm3) + ymm7
        vmovupd ymmword ptr [rcx + 8*rdi + 40], ymm3
        add     rdi, 8
        jmp     .LBB2_13
```

Flag	Effect
`-fno-math-errno`	Do not set `ERRNO` after calling any `math` functions that are executed in a single instruction. Since this program does not use any, it has no effect.
`-funsafe-math-optimizations`	Allow optimizations for floating-point arithmetic that (a) assume that arguments and results are valid and (b) may violate `IEEE` or `ANSI` standards. This flag enables: `-fno-signed-zeros`, `-fno-trapping-math`, `-fassociative-math` and `-freciprocal-math`.
`-ffinite-math-only`	Assume that results and arguments to floating point operations are neither `NaN` nor `±Inf`, allowing for various optimisations.
`-fno-rounding-math`	This is a default flag value that enables optimisations which assume default floating point behaviour when rounding.
`-fno-signaling-nans`	Similar to `-fno-trapping-math` assume that NaN values cannot generate user-visible traps.
`-fcx-limited-range`	Specify that a range-reduction step is not necessary when performing complex division. Since the program does not contain complex numbers, this has no effect.
`-fno-signed-zeros`	Ignore the sign of zeroes in operations. `IEEE` specifies that `+0.0` and `-0.0` have distinct behaviour, which prevents optimisations of some expressions from taking place.
`-fno-trapping-math`	Similar to `-fno-signaling-nans` assume that floating-point operations cannot generate user-visible traps, e.g. division by zero or over- or underflow.
`-fassociative-math`	Allow re-association of operands in floating point operations. The changes made are accurate mathematically but will not necessarily produce the same result with finite-precision floating point numbers. Since the program only outputs to five significant figures, this does not affect the results.
`-freciprocal-math`	Allow a reciprocal value to be used instead of division. This is mostly irrelevant in later iterations of the program where division has already been eliminated, but is useful when subexpressions can be eliminated or factorised.

Table II. Flags and effects enabled by `-ffast-math`.

[1] gcc(1) - linux man page. URL https://linux.die.net/man/1/gcc.

Index of Change	Execution Time / s											
	`gcc`				`clang`				`icc`			
	`-O0`	`-O1`	`-O2`	`-O3`	`-O0`	`-O1`	`-O2`	`-O3`	`-O0`	`-O1`	`-O2`	`-O3`
0 (no changes)	46	7.0	5.5	5.5	45	6.1	5.5	5.5	72	61	3.6	3.6
1	23	7.0	5.5	5.5	23	6.1	5.5	5.5	22	6.3	3.6	3.6
2	19	4.6	3.3	3.3	19	3.7	3.5	3.4	18	4.2	3.5	3.6
3	17	4.6	3.3	3.3	17	3.8	3.5	3.4	16	3.9	3.2	3.3
4	16	4.6	3.3	3.3	15	3.8	3.5	3.4	15	3.5	3.2	3.2
5	13	3.4	2.6	2.6	13	2.6	2.6	2.6	14	2.4	2.2	2.2
6	13	3.4	2.6	2.6	12	2.6	2.6	2.6	13	2.4	2.2	2.2
7	12	2.8	2.6	1.4	11	2.6	1.4	1.4	12	2.2	2.2	2.2
8	12	2.8	2.6	1.4	10	2.6	1.4	1.4	12	2.2	2.2	2.2
9	9.3	2.8	2.6	1.4	8.3	2.6	1.4	1.4	9.6	2.2	2.2	2.2
10	9.3	2.8	2.6	1.4	8.3	2.6	1.4	1.4	9.6	2.2	2.2	2.2
11	9.3	2.3	1.9	1.0	8.3	1.9	1.0	1.0	9.6	2.2	2.2	2.2
12	9.3	2.3	1.9	0.72	8.3	1.9	0.69	0.69	9.6	2.2	2.2	2.2
13	9.3	2.2	1.9	0.72	8.3	1.9	0.69	0.69	9.6	2.2	2.2	2.2
14	9.3	2.2	1.5	0.79[a]	8.3	1.3	0.62	0.62	9.6	1.5	1.4	1.4
15	10	2.4	1.5	0.72	9.0	1.3	0.55	0.55	10	1.5	1.4	1.5
16	9.3	2.4	1.5	0.72	8.3	1.3	0.55	0.55	9.6	1.5	1.4	1.5
17	8.8	2.4	1.5	0.72	8.2	1.4	0.55	0.55	8.9	1.5	1.5	1.5

[a] Disabling this compiler flag did not affect later results.

Table III. Average execution times for each compiler and optimisation level by change applied. Values were measured on `stan1` at a low-activity period. All runs were on a 256-by-256 grid for 10,000 steps.

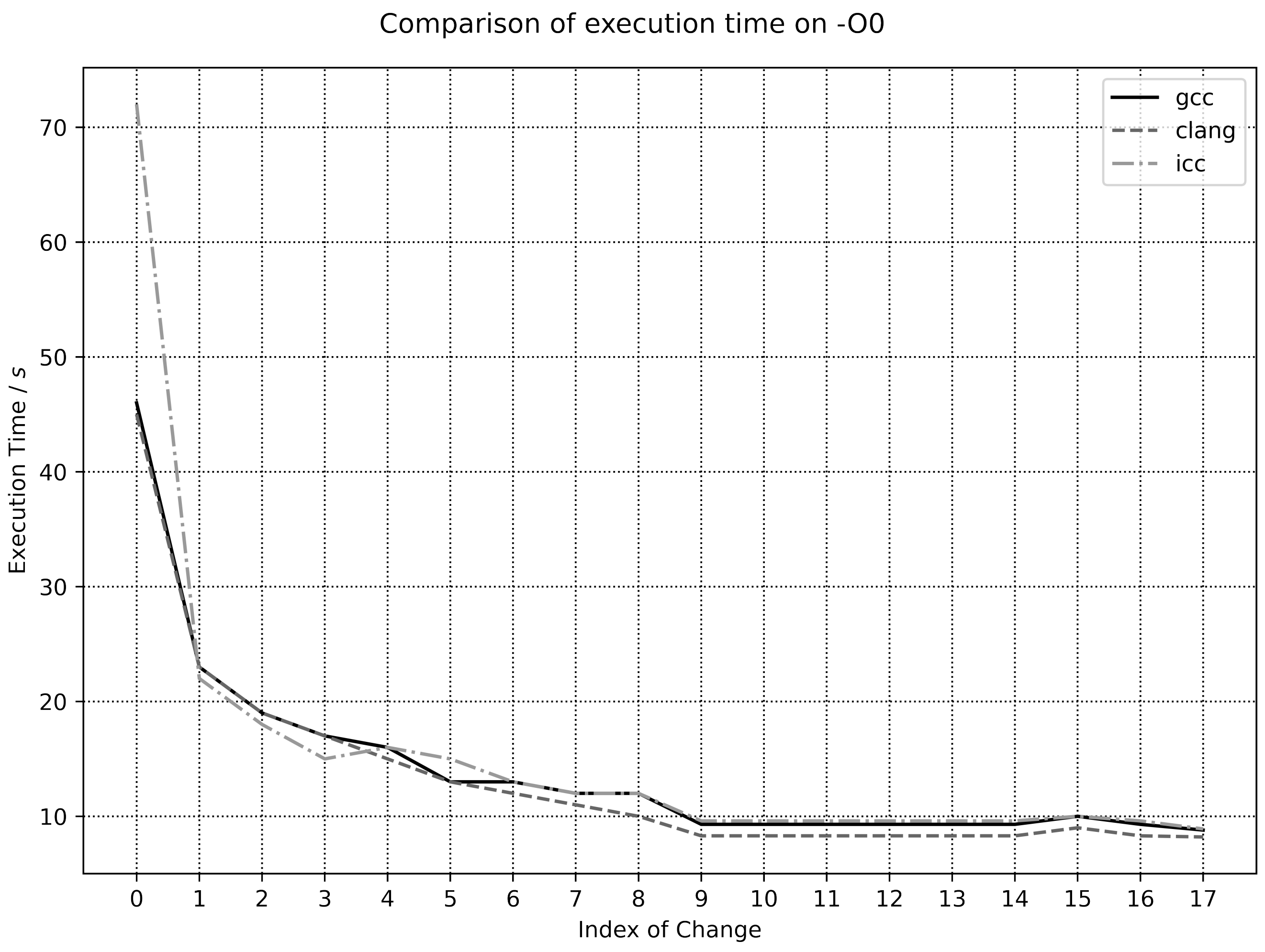

Figure 1. `-O0` data for each compiler tested.

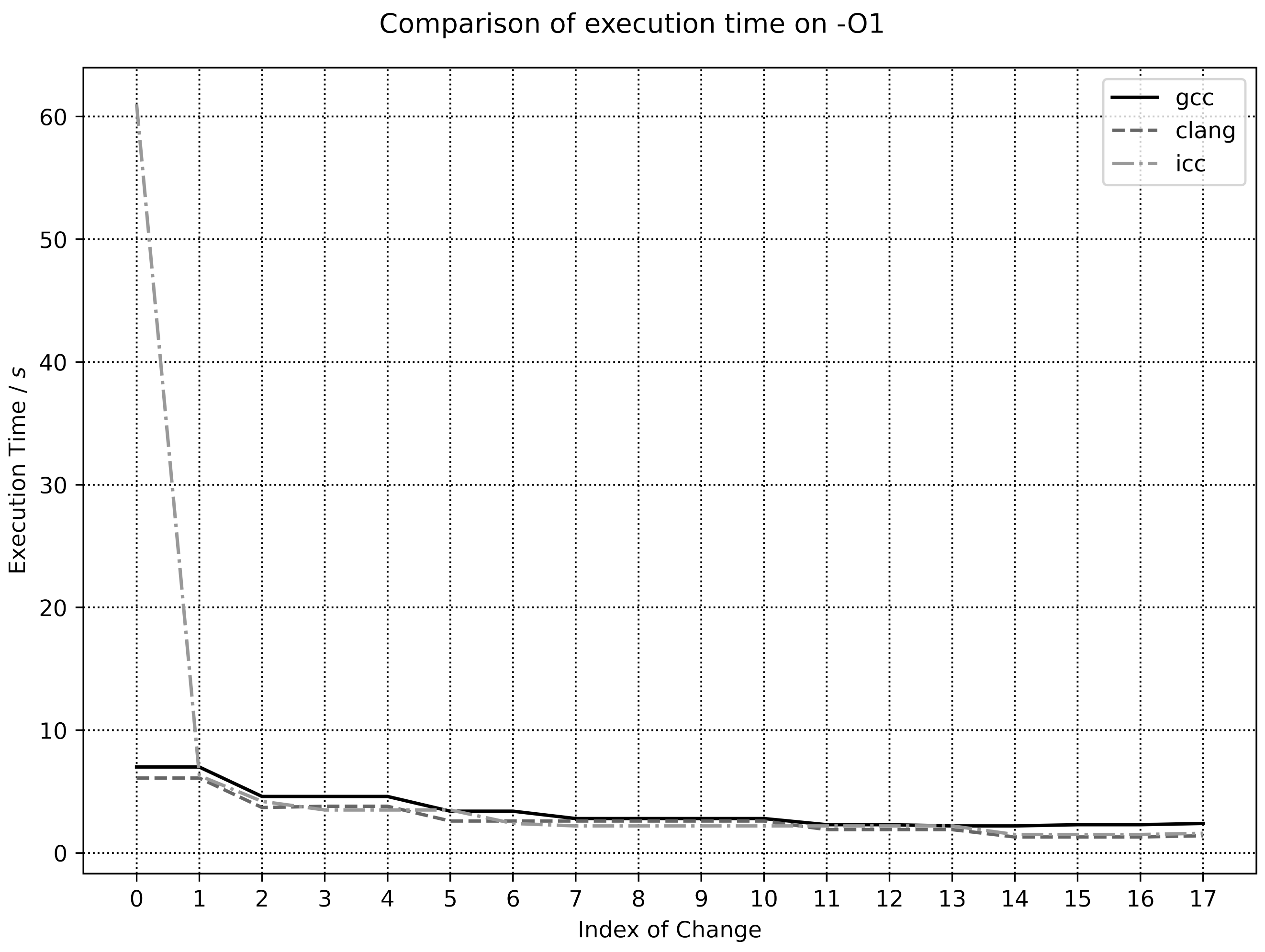

Figure 2. `-O1` data for each compiler tested.

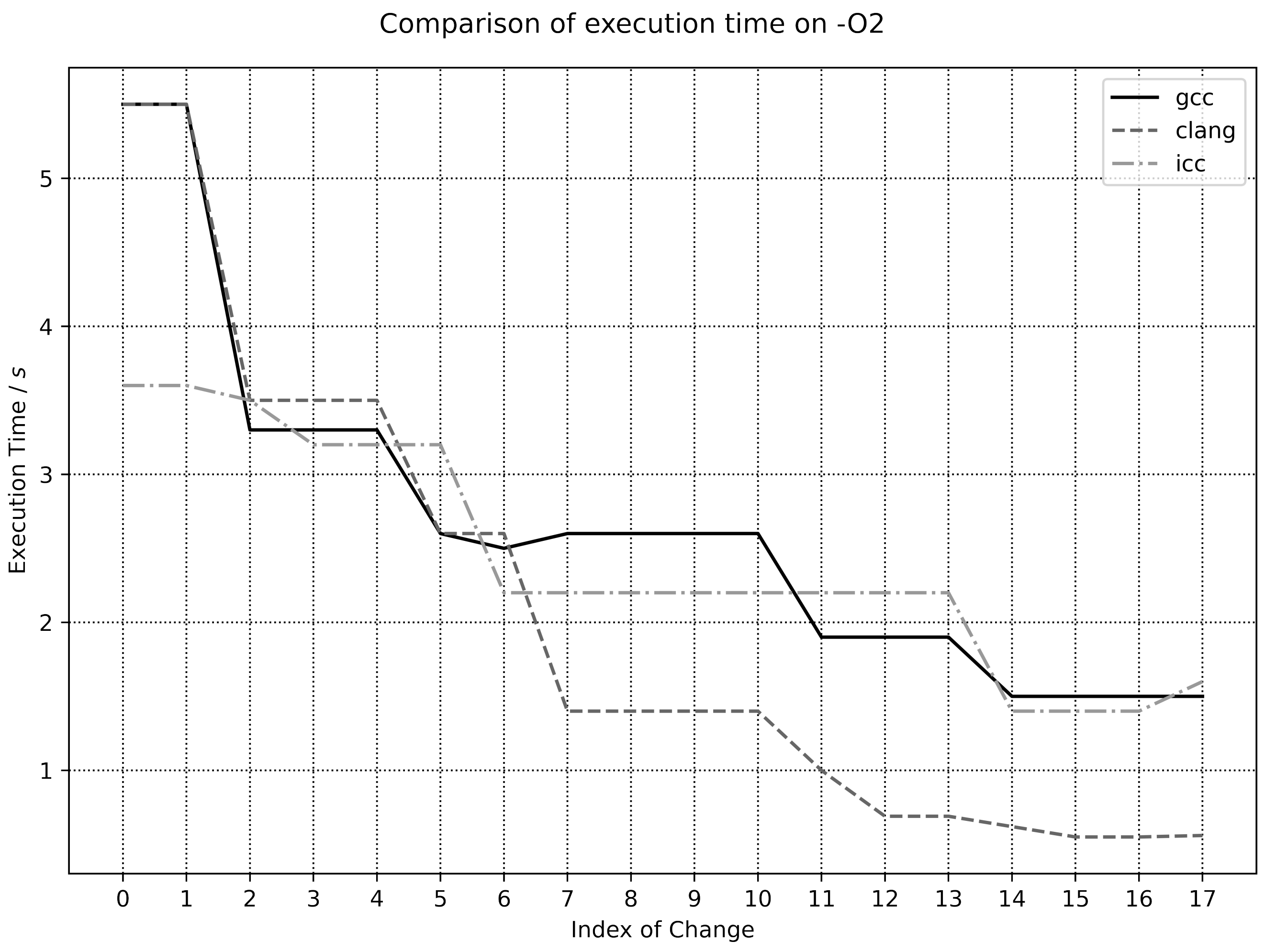

Figure 3. `-O2` data for each compiler tested.

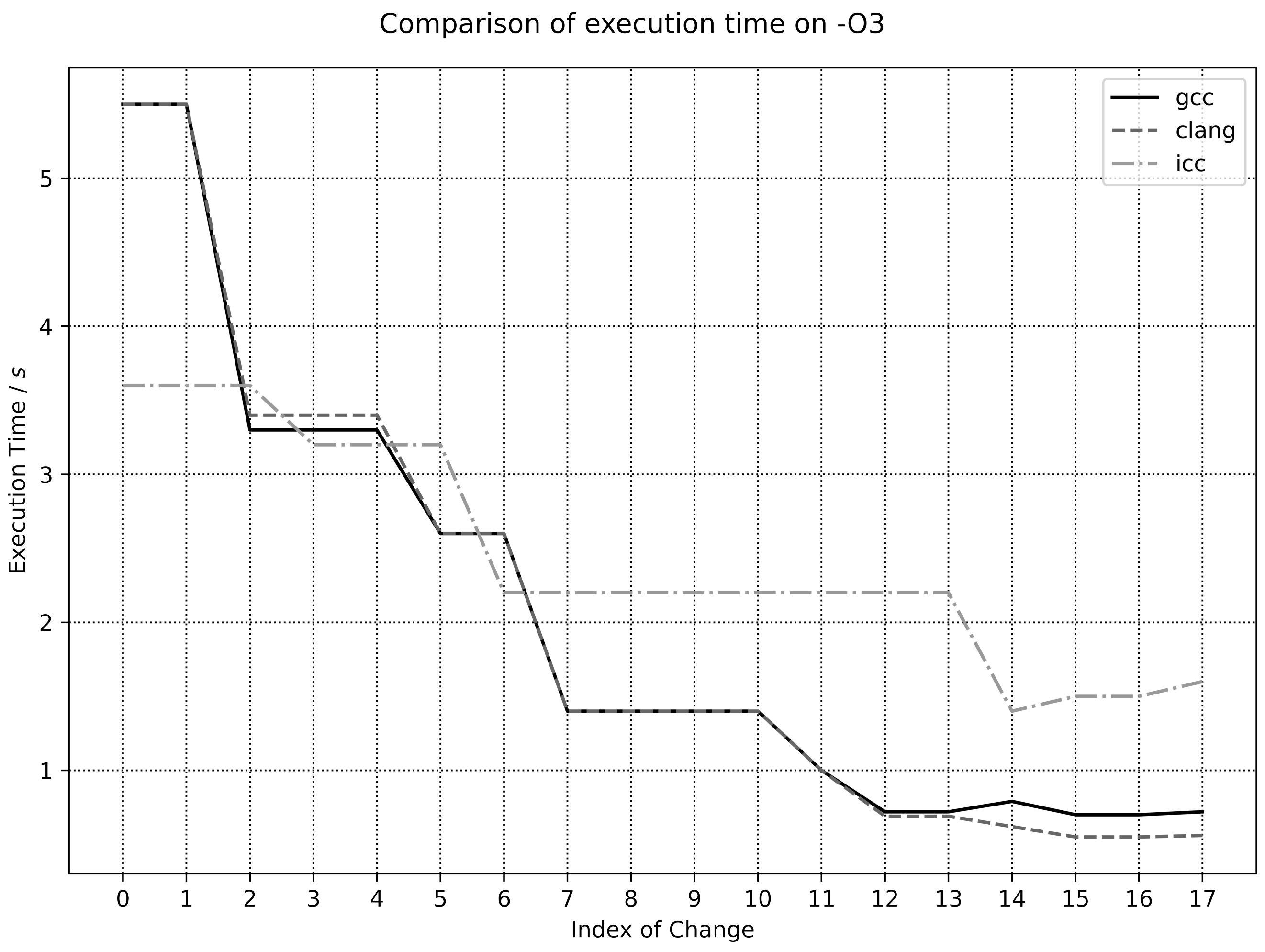

Figure 4. `-O3` data for each compiler tested.