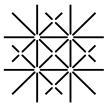

University of Basel

Verification and Validation of the OpenMP Standard Functionality with Focus on Scheduling Clauses

Bachelor Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
HPC research group
https://hpc.dmi.unibas.ch/

Advisor: Prof. Dr. Florina Ciorba
Supervisor: Dr. Jonas H. Müller Korndörfer

Sascha F. Maibach
sascha.maibach@unibas.ch
2019-932-441

2024-01-20

Abstract

This thesis discussed the implementation of a testsuite for the OpenMP scheduling clauses. Although there has been a big effort to test OpenMP functionalities implementation for different compilers, the schedule clauses remain untested. With a set of test templates, as well as a framework to generate specific test cases, this work aims to check any scheduling option against the OpenMP specifications of versions 4.5, 5.0, 5.1 and 5.2 for the 3 supported programming languages C, C++ and fortran. To provide an easy to use test suite, all automation regarding test generation, slurm, plotting and verifying has been implemented in a set of simple python scripts. As an initial effort in verifying scheduling clauses, 5 common compiler classes have been tested: GNU, LLVM, NVIDIA HPC, Intel classic and Intel OneAPI.

Table of Contents

1 Introduction

OpenMP is widely used as a convenient way to develop parallel applications. Since the first version, OpenMP 1.0 in 1997, there have been 5 Major versions, each adding more functionality and support for the newest language standards.
With each release, compiler vendors need to make sure to implement the new functionality in their products. What is still missing for this task, is a test suite to check if everything works as expected. Although OpenMP Validation and Verification (OMPVV) provides a framework and a lot of tests, the OpenMP schedule clauses remain untested.
This thesis aims to implement a set of tests to check the standard functionality of schedule clauses according to specifications, as well as a framework to generate, run, and plot those tests.
The secondary goal is to cover the tests with all supported languages, e.g. C++, C and Fortran 90. Beside this, all tests are supposed to run unmodified on every common compiler. Results shall be displayed in a table, as well as in plots for each dedicated test.
The specifications for schedule clauses remained unchanged at least since OpenMP version 4.5, therefore all tests will check conformity with versions 4.5, 5.0, 5.1 and 5.2.

2
Background

2.1 OmpVV

As part of the Exascale Computing Project, a team from the Oak Ridge National Lab and the University of Delaware started working on a framework that allows to check compiler implementations for their conformity with the OpenMP specifications. This framework is still under development, even after the Exascale Computing Project has ended. Most functions of the OpenMP API are tested and checked if they conform to the 4.5 and 5.0 specifications. However, testing of schedule clauses are still missing.

2.1.1 Existing tests

The OMPVV test suite consists of 731 [1] tests in total, of which 249 are written in fortran 90, 42 in c++ and 481 in c.

2.1.2 Structure

The project consists of a set of tests, common header files for reporting results and some framework to display the results as csv, json or HTML Prettified format.

The display methods allow a depiction of the results as yes/no table, but lack the possibility to plot any results.

[1] https://crpl.cis.udel.edu/ompvv/results/, Jan-17-2025

3
Related work

To the best of our knowledge, there is no similar public efforts reporting on the validation of OpenMP schedule clauses on compilers for OpenMP versions 4.5 - 5.2. However, scheduling has been tested for OpenMP 3.1 [10] in 2012, which might be outdated by now. But it shows that verifying implementations of the OpenMP standard have been around for quite a while.

While there are no current tests for scheduling clauses, several papers have been published that work in improving the verification of the OpenMP specifications [7] [8] [9] [11].

Some of them, such as Sun et al. [9], use advanced techniques like bounded models checking to verify whole programs. Our work sets the focus on Unit-tests by checking just one very simple schedule clause. While this alone does not verify whole codebases but a full fledged test suite like OMPVV has the power to show that every single piece of the OpenMP API does itself work as intended. This certainty makes checking whole parallel programs much easier, as we already know that the basics work, and can focus on high-level problems.

4

Testing OpenMP schedule clauses

4.1 Overview

The basic idea of this test suite is to run the most basic loop with specified OpenMP scheduling clauses. For each iteration, the ID of the executing thread is saved in an array.
After the test, this array is aggregated, meaning that the consecutive iterations of the same thread are summed up. This yields a database with entries of the form [threadID-iterations]. All data is saved in a CSV file for further processing. This includes plotting and checking the results for pass/fail. Every test is repeated 10 times to detect and avoid outliers.

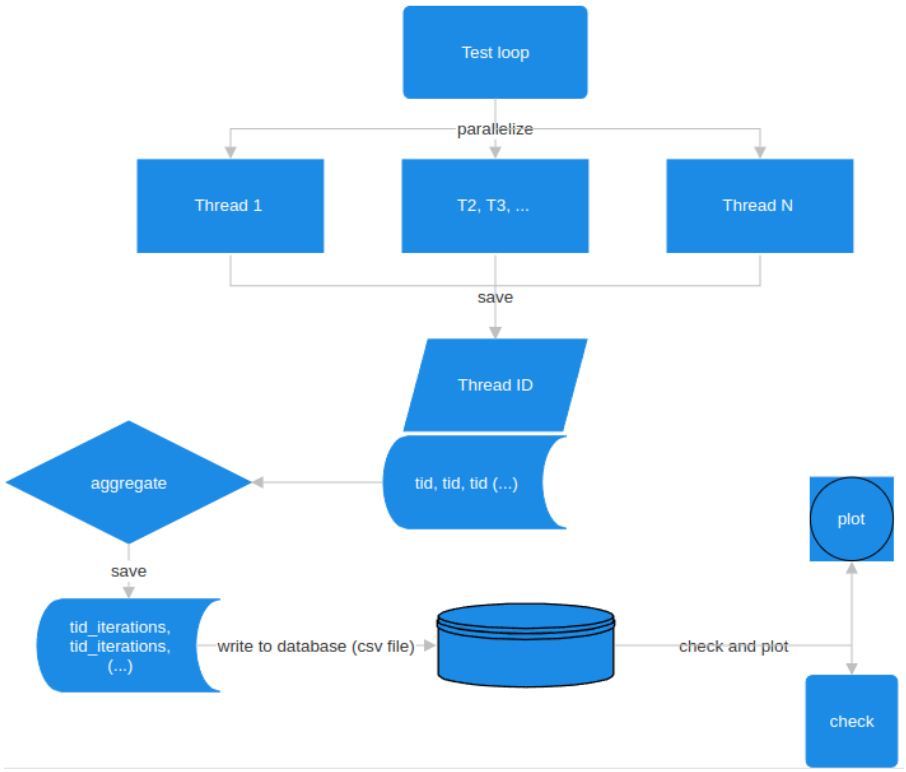

Figure 4.1: Overview of test procedure

4.2 Data collection

The content of a test loop consists of a single writing operation. An array with length [iterations] is given to the loop and written by the executing thread in each iteration. Because each thread writes at position [iteration], there is no race condition. The array holds [iterations] int32 elements, which is $2 * 10^9$ for the standard test. To save on space, the results are aggregated in a way that assigns each thread consecutively executed iterations, known as chunks.

```
// Aggregation function in C/C++
void aggregate(const int* array, char** aggregated_string_array, int n) {
  int tid = array[0];
  int index = 0;
  int n_chunks = 1;
  const char format[] = "%d-%d";
  for (int i = 1; i < n; i++) {
    if (array[i] == array[i-1]) {
      n_chunks++;
    } else {
      snprintf(aggregated_string_array[index], 15, "%d-%d", tid,n_chunks);
      n_chunks = 1;
      tid = array[i];
      index++;
    }
  }
  sprintf(aggregated_string_array[index],format,tid,n_chunks);
}
```

```
! Aggregation function in fortran 90
SUBROUTINE aggregate (int_array, char_array, len_array)
    IMPLICIT NONE
    INTEGER, INTENT(IN) :: int_array(:)
    CHARACTER(12), DIMENSION(len_array) :: char_array
    INTEGER, INTENT(IN) :: len_array
    INTEGER :: tid, index, n_chunks, i
    tid = int_array(1)
    index = 1
    n_chunks = 1
    DO i = 2, len_array
      IF (int_array(i) == int_array(i-1)) THEN
        n_chunks = n_chunks + 1
      ELSE
        WRITE(char_array(index), '( i0, a, i0 )') tid, "-", n_chunks
        n_chunks = 1
        tid = int_array(i)
```

```
        index = index + 1
      END IF
    END DO
    WRITE(char_array(index), '( i0, a, i0 )') tid, "-", n_chunks
END SUBROUTINE aggregate
```

4.3 Test code

The very basic *parallel for* loops used for testing resemble the ones used in the official OpenMP examples [2] [4].
Parameters for a data array, thread count and iteration count have been added to parametrize the tests. Thread count and iteration count are read-only, ergo they do not pose any implication. The data array is written by every thread, but every iterations writes in its own memory, that is why there is no race condition.

4.3.1 C/C++

The test loop has been written such that it can be compiled as C and C++ code. Due to the simplicity of the loop there is no difference between the two languages.

4.3.1.1 Simple loop

A simple for loop is tested with $2 * 10^9$ iterations. The code looks as follows:

```
/*
* result_array: array of length iterations to store data
* n_threads: amount of threads to be assigned to parallel region
* iterations: amount of iterations to run
*/
int run_test_loop(int *result_array, int n_threads, const int iterations) {
    int i;
    #pragma omp parallel for schedule(SCHEDULE) num_threads(n_threads)
    for (i = 0; i < iterations; i++) {
        result_array[i] = omp_get_thread_num();
    }
    return 0;
}
```

4.3.1.2 Collapsed nested for loop

A nested for loop is tested with the collapse keyword. This collapses the 2 loops into one larger iteration space that is then divided according to the schedule clause. Each loop has $\sqrt{2 * 10^9}$ iterations.

```
/*
* result_array: array of length iterations to store data
* n_threads: amount of threads to be assigned to parallel region
* iterations: amount of iterations per loop
*/
int run_test_loop(int* result_array, int n_threads, const int iterations) {
    int i,j;
    #pragma omp parallel for collapse(2) schedule(SCHEDULE) num_threads(n_threads)
    for (i = 0; i < iterations; i++) {
        for (j = 0; j < iterations; j++) {
        result_array[i*iterations + j] = omp_get_thread_num();
        }
    }
  return 0;
}
```

4.3.2 Fortran

All fortran code is written according to the f90 standard.

4.3.2.1 Simple loop

Same as for C/C++, the fortran test uses a very simple loop with the minimal !$omp statments

```
! simple fortran test loop
    !$OMP PARALLEL DO schedule(INSERT_SCHEDULE_HERE)
    DO j=1, iterations
      result_array(j) = OMP_GET_THREAD_NUM()
    END DO
    !$OMP END PARALLEL DO
```

4.3.2.2 Collapsed nested for loop

The fortran nested loop with *collapse(2)* statement This collapses the 2 loops into one larger iteration space that is then divided according to the schedule clause. Each loop has $\sqrt{2 * 10^9}$ iterations.

```
! collapse(2) nested fortran test loop
    !$OMP PARALLEL DO schedule(INSERT_SCHEDULE_HERE) collapse(2)
    DO j=1, iterations
      DO i=1, iterations
        result_array((j-1)*iterations+i) = OMP_GET_THREAD_NUM()
        END DO
    END DO
    !$OMP END PARALLEL DO
```

4.4 Test automation

Because schedule kinds cannot simply be parametrized, this test suite works with templates that are copied and adjusted on demand. For each loop kind (simple loop, nested loop) and language (C/C++, Fortran) there is a test template. On generation, the template is copied and the schedule kind is written in the OMP loop header.
Each test then compiles the schedule template to its own object files and runs them accordingly.

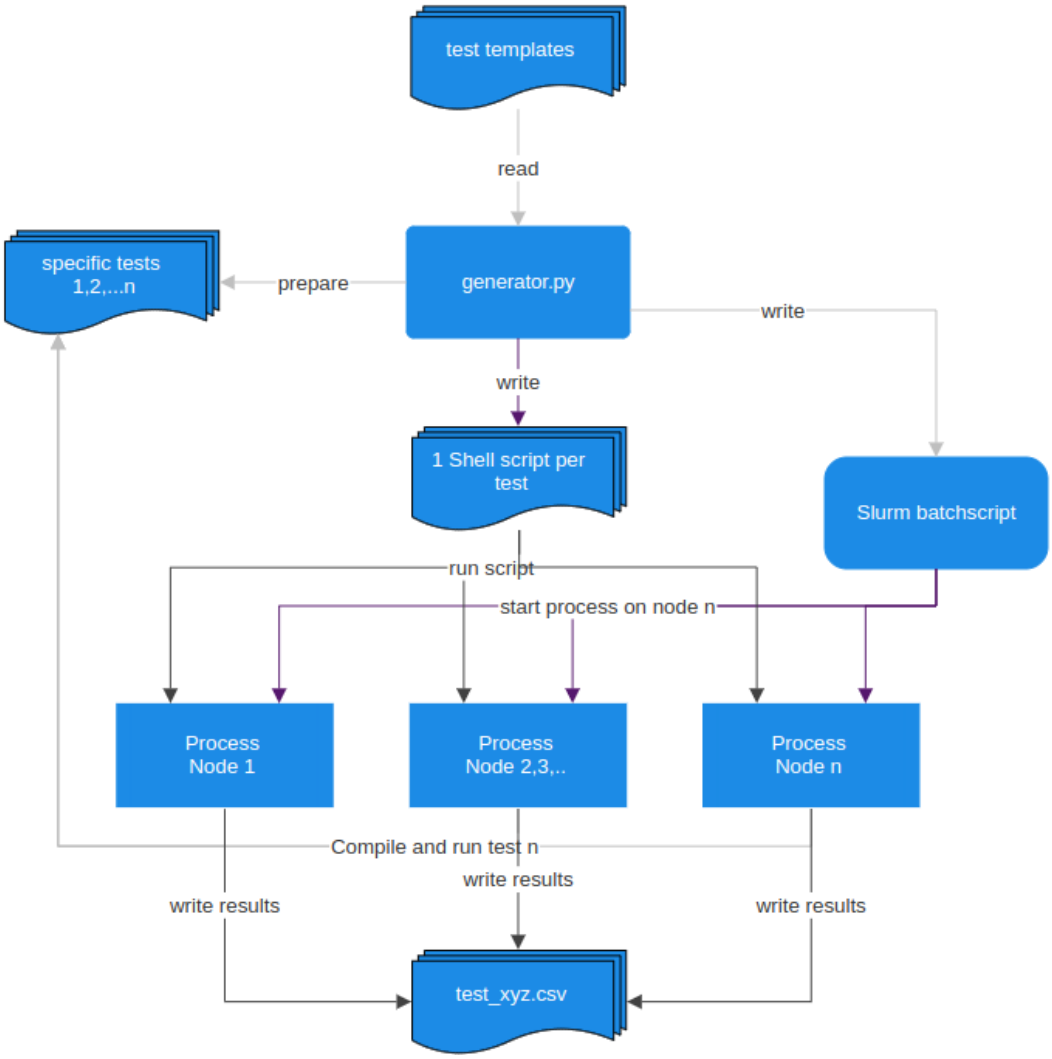

Figure 4.2: Overview of test generation

4.4.1 Test code generation

The basic loop to generate all the tests looks as follows:

```
for loop in LOOP_KINDS:
    for vendor in VENDORS:
        for schedule in SCHEDULES:
            for flag in OPTIMIZATION_FLAGS:
                for compiler in get_compilers_by_vendor(vendor):
```

In every iteration a test with the pattern *vendor_compiler_schedule_flag_loop* is generated. For each test, the following actions are executed:

1. Generate the test code
   Each test needs a specific schedule. The test template is copied, named accordingly and the schedule kind is written in the omp loop section.

2. Generate helper files
   In Fortran files, the module name of the test file and the helper file need to match. Because we want to compile every test separatly, including the helper file, the module needs to be renamed for each test.

3. Writing the bash script for slurm execution
   For each test, a dedicated bash script is written which runs everything needed for a test, including:

   - Loading the module needed for the compiler
   - Compiling the helper script
   - Compiling and linking the test code
   - Running the test
   - Unloading the module

4. Adding the bash script to a file
   In the end, every bash script is run as job by calling *srun bash scriptname.bash* from the initial slurm array job.

In the end, the slurm job script is generated, as described in the next section.

4.4.2 Slurm

All the tests can be run as an array job via slurm. The idea is to write one bash script for each test run as described above, and call those from a master script.
The batchscript run by *sbatch* calls the tests as follows:

```
(..)
#SBATCH --array=1-750%100
$(head -$SLURM_ARRAY_TASK_ID runfile.cmd | tail -1)
```

with the *runfile.cmd* looking like this:

```
(..)
srun bash test1.bash
srun bash test2.bash
(..)
```

4.5 Result storage

The results from each tests are stored in a CSV file, named accordingly. The total amount of data from running all tests sums up to approximatly 35 GB.
This number has been drastically reduced by the introduction of the *aggregate* function. The raw data from the tests would consume $2 * 10^9 entries * 4bytes/entry * 10repetitions \approx 80GB/test$.

4.5.1 CSV format

The generated CSV file from each test has the following format:
Compiler,Number of threads,total iterations,thread id-iterations count
with data types
String,int,int,String as (int32-int32),(..)
Example for default static scheduling:

$$gcc, 20, 2000000000, 0 - 50000000, 1 - 50000000, ..., 19 - 50000000$$

4.6 Verification of results

The script responsible for checking the results is located in *data_analyser.py*, written in Python. To speed up the code, I/O operations have been reduced to a minimum of reading each file once:

```
entries_list = []
with open(file, 'r') as csvfile:
    for line in csvfile:
        comma_before = find_first_data_entry(line)
        entries = []
        comma_after = line.find(',', comma_before+1)
        while comma_after != -1:
            entries.append(line[comma_before+1:comma_after])
            comma_before = comma_after
            comma_after = line.find(',', comma_before+1)
        entries.append(line[comma_before+1:])
        entries_list.append(entries)
return entries_list
```

4.6.1 Static verification

Static scheduling without a specified chunk size should yield chunks with size $\frac{iterations}{threadcount}$ chunks. If the iteration count is not divisible without remainder, some chunks will have 1 more iterations, e.g. static scheduling is guaranteed

$$iff\ abs(max(chunk) - min(chunk) <= 1$$

For completness, the sum of all iterations is also checked against the logged iteration count from the data.

```
for chunk in chunk_list:
    if abs(first_chunk_size - chunk) > 1:
        passed -= 1
passed += 1
```

4.6.2 Verification of static or dynamic schedule with chunk size 1000

Static or dynamic scheduling with chunk size 1000 is expected to distribute chunks with 1000 iterations each to every thread. For static scheduling this happens in a round-robbin manner, with dynamic scheduling it might happen that 2 consecutive chunks are assigned to the same thread. The verification step looks the same: First check the total iterations against the logged iteration count. Then check if every chunk is 1000 or a multiple of 1000. The last might hold fewer iterations if the total iteration count is not divisable by 1000.

```
for chunk in chunk_list:
    if chunk != 1000 and not chunk == chunk_list[-1] and not chunk % 1000 == 0:
        passed -= 1
passed += 1
```

4.6.3 Verification of guided scheduling

Now the verification of a list of chunks from guided scheduling is somewhat more difficult than static or dynamic. What we did here is precalculate the expected chunk sizes from the specifiactions, and compare the result to this list. The expected chunk sizes are calculates as:

$$size(chunk) = max(\frac{total\ iterations\ left}{amount\ of\ threads}, 1)$$

The points of this curve are calculated in Python as:

```
while iterations > 0:
        chunk_size = int(math.floor(iterations / n_threads))
        chunk_sizes.append(chunk_size) if chunk_size > 0 else 1
        iterations -= chunk_sizes[-1]
```

For the full verification we allowed some freedom of implementation, meaning that every chunk within 10% of the expected value is allowed. Also it was nessecary to account for consecutive chunks assigned to the same thread. That is why the inner loop searches for

sums of consecutive chunks. The calculation of chunk sizes below 10 is highly different between compilers, so it was omitted if all larger chunks are correct.

```
i = 0
for chunk in chunk_list:
    if chunk < 10:
        break
    if abs(chunk - guided_chunks[i]) > math.ceil(int(chunk/10)):
        for n in range(2,50):
            if abs(chunk - sum(x for x in guided_chunks[i:i+n])) <
                        math.ceil(int(chunk/100)):
                i += n-1
                break
            else:
                if n >= 50:
                    passed -= 1 if chunk > 1000 else 0
    i += 1
passed += 1
```

4.6.4 Finding auto schedules

Auto scheduling means that any of the other schedules is chosen. The verifications used in this thesis are incomplete, as auto could theoratically choose something like *static, 850*. But it turned out that most of the time auto scheduling chooses default static or some kind of dynamic scheduling. To find out which is the case, we simply check the data from auto scheduling with every other verification test.

5 Results

For each test, there is a simple pass/fail check as well as 2 plots for advanced analysis. To avoid overfilling this section only exempts are provided here. All checks can be found in Appendix A.
For this tests the following compiler versions have been used:

Compiler	Version
gcc, g++, gfortran (GCC)	12.3.0
clang, clang++	10.0.0
nvc, nvc++, nvfortran	20.11-0
icc, icpc, ifort (ICC)	2021.10.0 20230609
ifx, icx Intel(R) oneAPI DPC++/C++ Compiler	2023.2.0
ifx (IFX)	2023.2.0 20230721

Table 5.1: Compiler versions used in tests

5.1 Static scheduling

Without a specified chunk size, the static schedule distributes 1 round of $\frac{iterations}{thread\ count}$ per thread. With $2 * 10^9$ iterations and 20 threads, each thread should receive $\frac{2*10^9}{20} = 10^8$ iterations. All results for this schedule clause are valid and passed the checks.

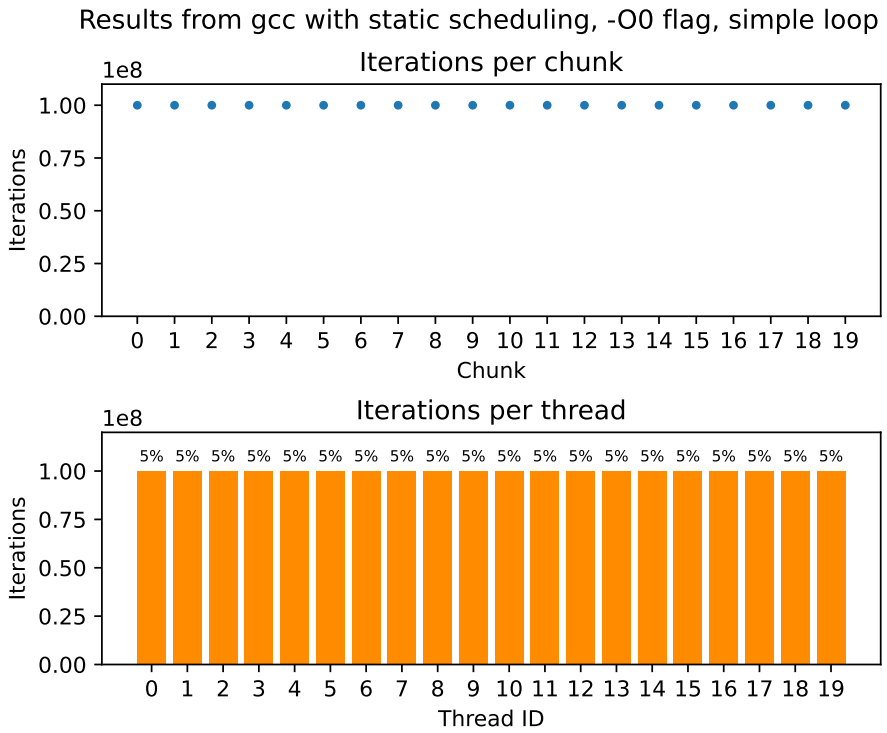

Figure 5.1: Example result of schedule(static) tests.

5.1.1 Static with chunksize 1000

For static scheduling with chunk size 1000, an evenly distributed load balance is expected. With $2 * 10^9$ iterations, $1 * 10^6$ chunks à 1000 iterations are expected.

The following figure shows clearly that the static scheduling for the specified parameters does work as expected.

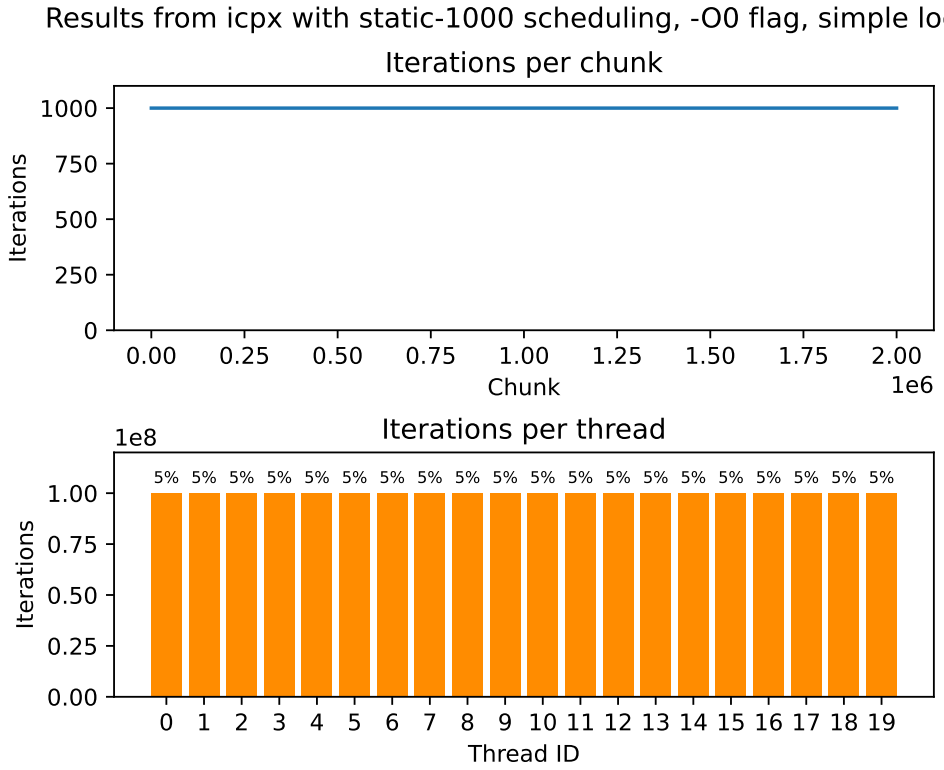

Figure 5.2: Example result of schedule(static,1000) tests.
Top: Iterations per chunk
Bottom: Iterations per thread

5.2 Dynamic scheduling

Dynamic scheduling is the wild west among the scheduling techniques. There will most likely be some load balancing, but you can never know what happens exactly.

All tests regarding dynamic scheduling passed, meaning that all compilers comply to the specifications in terms of dynamic scheduling with fixed chunk size. Due to huge data file sizes chunk size of 1000 has been chosen for this test.

As can be seen in the plots, the chunk size and load balance differs heavily between different optimization flags, even for the same compiler.

Because of the dynamic nature of the chunk distribution, a thread can receive several consecutive chunks. The verification method checks for multiples of 1000. For all compilers we could show that every chunk is divisible by 1000 without remainder, except the last chunk which takes the remaining iterations.

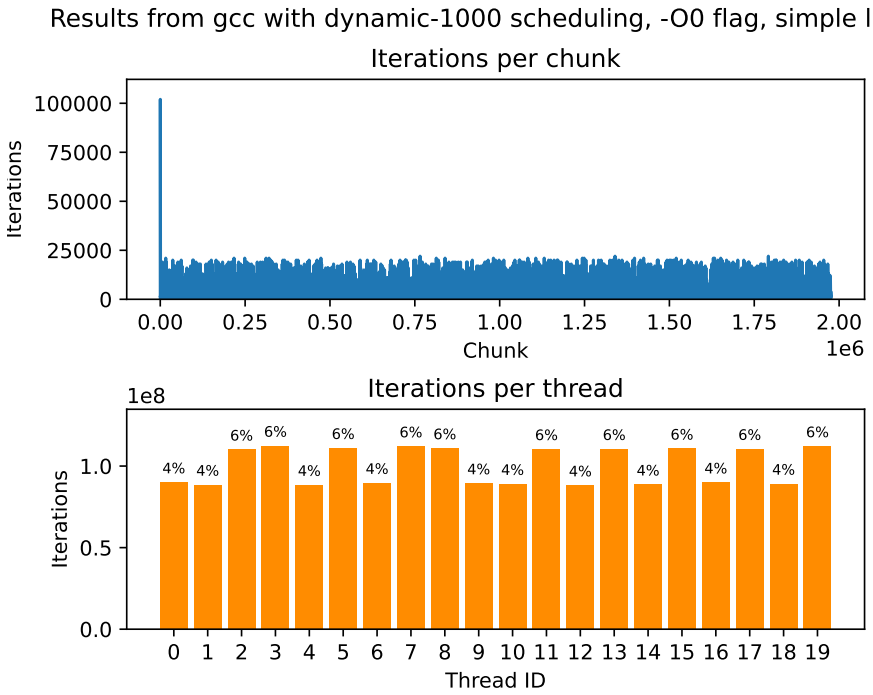

Figure 5.3: Example result 1 of schedule(dynamic,1000) tests.
In this case, 1 thread received a lot of chunks at the beginning. After that, the distribution was quite evenly.

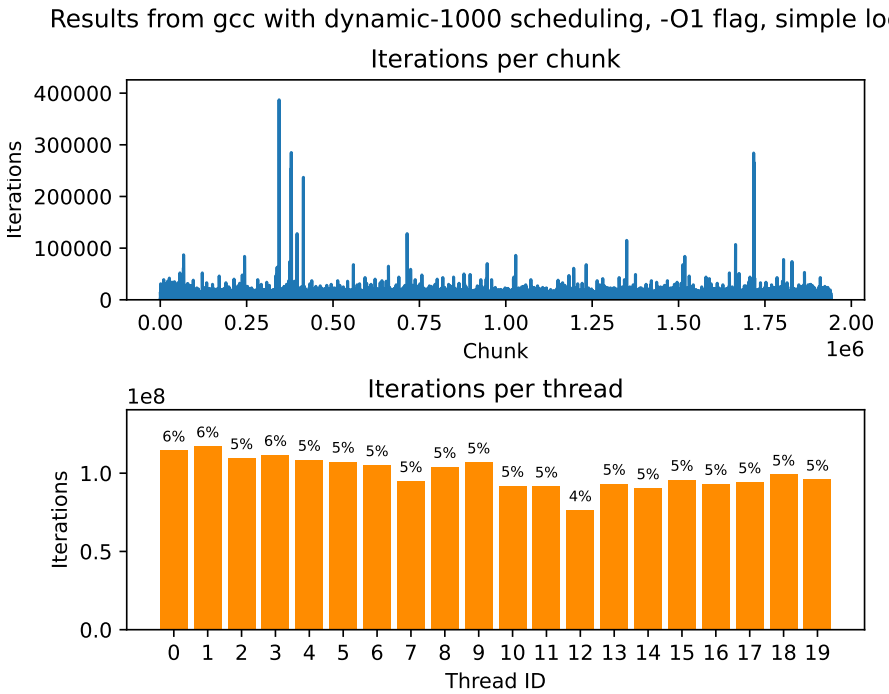

Figure 5.4: Example result 2 of schedule(dynamic,1000) tests.

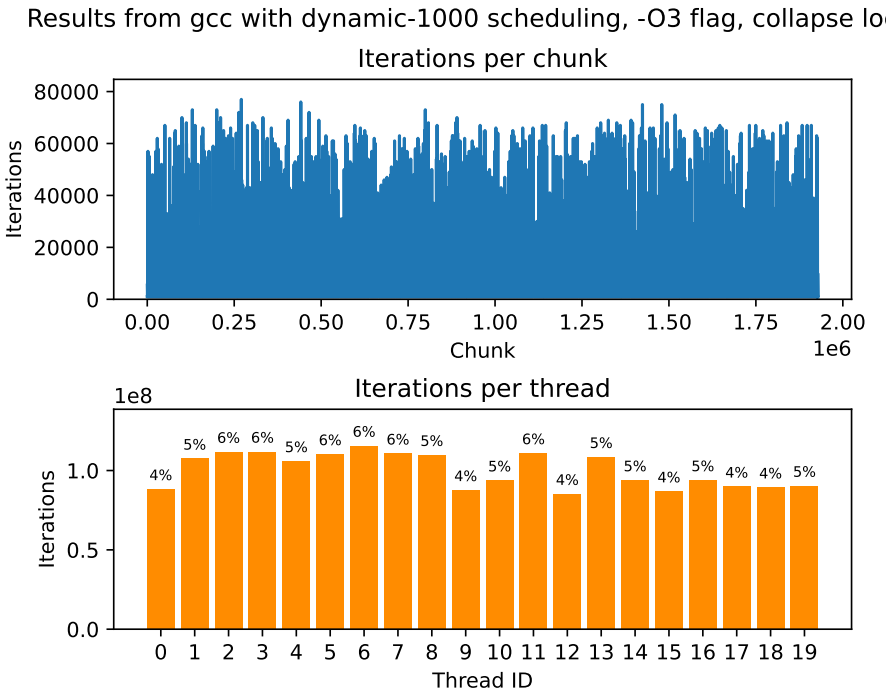

Figure 5.5: Example result 3 of schedule(dynamic,1000) tests.
Top: Iterations per chunk
Bottom: Iterations per thread

5.3 Auto scheduling

From 140 tests with auto scheduling, 60 have been identified as static scheduling with default chunk size and 80 as dynamically scheduled.

The static schedulers are the GNU and NVIDIA compilers. Dynamic scheduling is chosen by LLVM and Intel compilers.

5.4 Guided scheduling

Surprisingly, several guided test cases failed. Namely all tests with LLVM and Intel compilers.

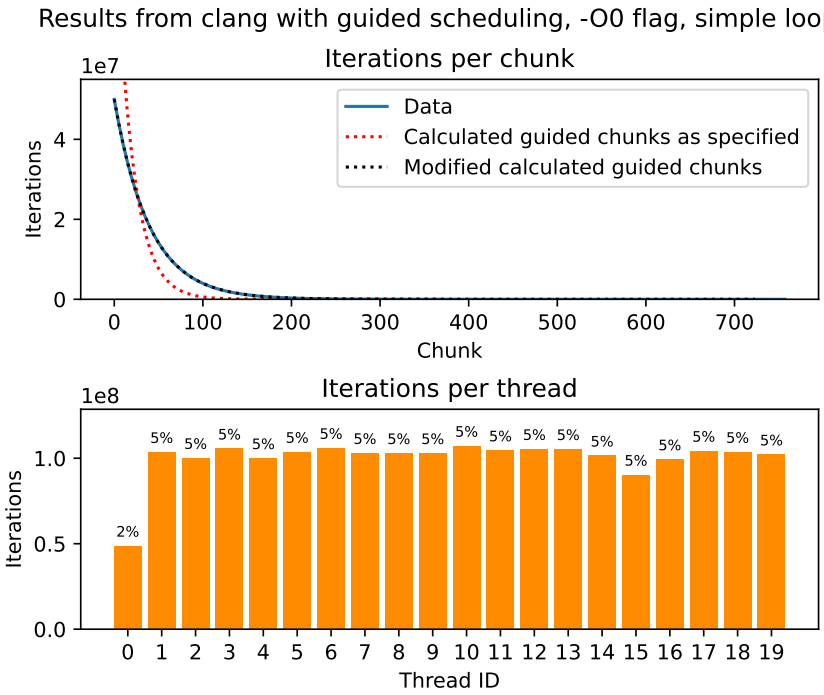

Figure 5.6: Example result of a failed guided test. The chunk sizes are clearly not as expected by specifiaction

5.5 Standard compliancy

In total 110 tests failed. Exactly those with guided schedule from LLVM, Nvidia and Intel compilers. As can be seen in the next chapter, the guided chunk sizes are calculated slightly different. That is why those tests have been marked as failed.

	GNU	LLVM	NVIDIA	Intel classic	Intel OneAPI
static default					
static, 1000					
dynamic, 1000					
auto					
guided					

Table 5.2: Standard compliancy results.
Red: tests failed
Green: All tests successfull

6 Analysis

6.1 Guided scheduling

"When kind is guided, the iterations are assigned to threads in the team in chunks. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned. For a chunksize of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to 1"
Guided scheduling definition, taken from official specifications: [1][3][5][6]

For guided scheduling, the implementation of the chunk size calculation differs between the compilers. The tests revealed that there are 2 major versions of the calculations: The specified and a slightly modified calculation.

$$specified \ : \ chunk\_size = \frac{remaining \ iterations}{number \ of \ threads}$$

$$modified \ : \ chunk\_size = \frac{remaining \ iterations}{\mathbf{2} \ * number \ of \ threads}$$

According to the results only the GNU compilers (gcc, g++, gfortran) use the specified chunk size calculations. LLVM compilers, as well as all derivates (Intel classic, Intel OneAPI, Nvidia), use the modified version.

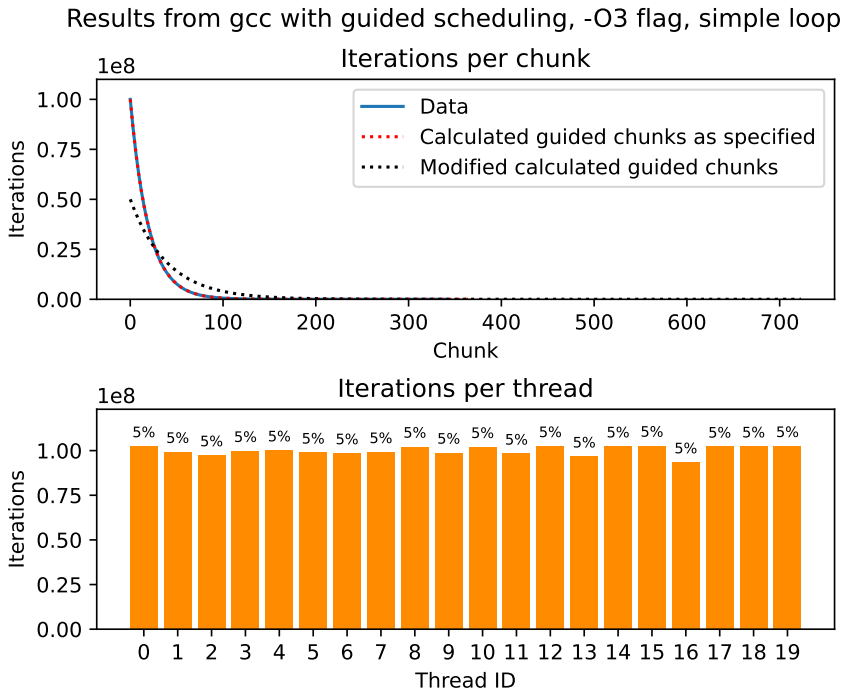

Figure 6.1: GCC test results, the chunk sizes clearly follow the expected decrease as specified

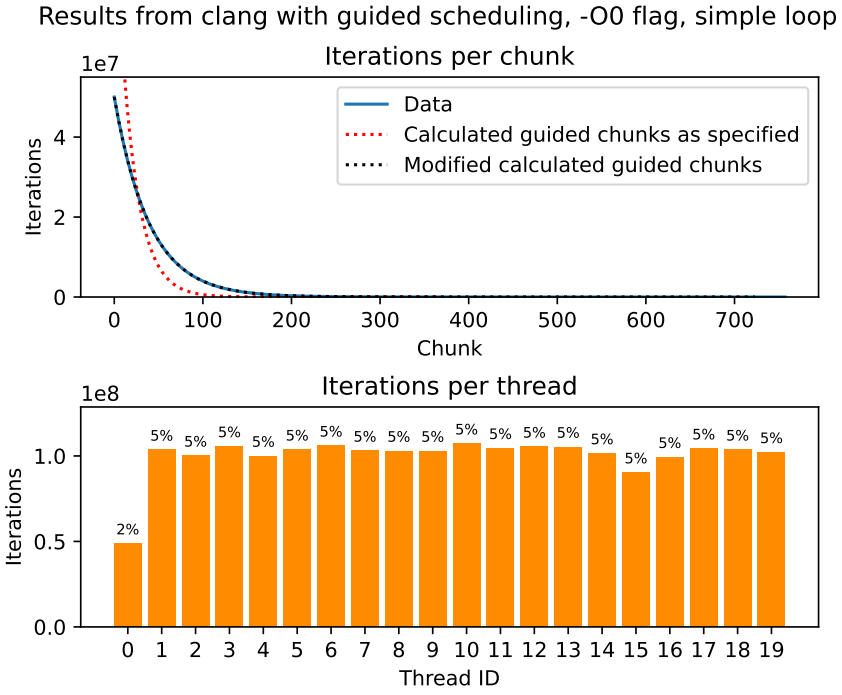

Figure 6.2: Clang results, the chunk sizes follow exactly the decrease of the modified size calculation

6.2 Static scheduling

When *schedule(static, chunk_size)* is specified, iterations are divided into chunks of size chunk_size, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number. When no chunk_size is specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread. The size of the chunks is unspecified in this case.

With the following 2 checks, we have shown that the implementations conform to the specification by scheduling chunks of 1000 for *schedule(static,1000)* and $\frac{iterations}{num\_threads}$ for *schedule(static)*

```
# Checking chunk sizes; pass a test if chunksize is 1000
for chunk in chunk_list:
    if chunk != 1000 and not chunk == chunk_list[-1]:
        passed -= 1
passed += 1
```

```
# Check that chunks sizes differ by a maximum of 1
for chunk in chunk_list:
    if abs(first_chunk_size - chunk) > 1:
        passed -= 1
passed += 1
```

6.3 Dynamic scheduling

When *schedule(dynamic, chunk_size)* is specified, the iterations are distributed to threads in the team in chunks. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed. Each chunk contains chunk_size iterations, except for the chunk that contains the sequentially last iteration, which may have fewer iterations.

This specification has been verified by evaluating each chunk and checking if it is either 1000 or a multiple of 1000 in case a thread gets consecutive chunks assigned.

```
for chunk in chunk_list:
    if chunk != 1000 and not chunk == chunk_list[-1] and not chunk % 1000 == 0:
        passed -= 1
passed += 1
```

6.4 Auto scheduling

When *schedule(auto)* is specified, the decision regarding scheduling is delegated to the compiler and/or runtime system. The programmer gives the implementation the freedom to choose any possible mapping of iterations to threads in the team.

With all other checking methods combined, we have a way to determine what schedule type is chosen by the compiler when auto scheduling is provided by the user.

7 Conclusion & outlook

7.1 Conclusion

There is a lot of work hidden in this task. Although the tests are quite simple, there are numerous bugs and inconsistencies that need fixing. Especially during compiling and linking, problems arise when using the same test code for different compilers. Deep knowledge of internal processes for each compiler come in handy when debugging manifold error messages. Except from the modified LLVM guided scheduling chunk size calculations, every test passed 10 times in a row. The fact that such an inconsitency has been found, shows that the tests are working and nessecary.
In conclusion we can say that GNU compilers conform, in terms of scheduling, to the OpenMP specifiactions of version 4.5, 5.0, 5.1 and 5.2. LLVM and derivated compilers (NVIDIA and Intel) use a special guided schedule, which is not exactly as specified. It does, however, still work, although not as intended.
The Python framework written for this thesis was probably not strictly necessary, but very useful for plotting and data processing.

7.2 Outlook

This thesis presents a solid foundation for testing OpenMP schedule clauses. There are several aspects that can be improved and implemented.
The tests include the most common free compilers, GNU and LLVM, as well as the vendor specific compilers from NVIDIA and Intel. Additional compilers used in important systems may be tested, such as Cray compilers or cc/CC.
With scheduling tests ready and tested, a pull request to the OMPVV project seems logical. Some modifications are needed to conform to this framework.
OpenMP version 6.0 was released during the work on this thesis. At first glance, the scheduling did not change that much, the tests should work with that version as well.

Bibliography

[1] OpenMP Architecture Review Board. Openmp 5.0 application programming interface. Technical report, OpenMP, november 2015. URL https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[2] OpenMP Architecture Review Board. Openmp 4.5 application programming interface - examples. Technical report, OpenMP, November 2016. URL https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf.

[3] OpenMP Architecture Review Board. Openmp 5.0 application programming interface. Technical report, OpenMP, november 2018. URL https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[4] OpenMP Architecture Review Board. Openmp 5.0 application programming interface - examples. Technical report, OpenMP, june 2020. URL https://www.openmp.org/wp-content/uploads/openmp-examples-5-0-1.pdf.

[5] OpenMP Architecture Review Board. Openmp 5.0 application programming interface. Technical report, OpenMP, november 2020. URL https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf.

[6] OpenMP Architecture Review Board. Openmp 5.0 application programming interface. Technical report, OpenMP, november 2021. URL https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

[7] Jose Monsalve Diaz, Swaroop Pophale, Oscar Hernandez, David E. Bernholdt, and Sunita Chandrasekaran. Openmp 4.5 validation and verification suite for device offload. In Bronis R. de Supinski, Pedro Valero-Lara, Xavier Martorell, Sergi Mateo Bellido, and Jesus Labarta, editors, *Evolving OpenMP for Evolving Architectures*, pages 82–95, Cham, 2018. Springer International Publishing. ISBN 978-3-319-98521-3.

[8] Thomas Huber, Swaroop Pophale, Nolan Baker, Michael Carr, Nikhil Rao, Jaydon Reap, Kristina Holsapple, Joshua Hoke Davis, Tobias Burnus, Seyong Lee, David E. Bernholdt, and Sunita Chandrasekaran. Ecp sollve: Validation and verification testsuite status update and compiler insight for openmp. In *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 123–135, 2022. doi: 10.1109/P3HPC56579.2022.00017.

[9] Liang Sun, Bailin Lu, Liangze Yin, Zhe Bu, and Wenjing Jin. Openmp program verification based on bounded model checking. In *2023 IEEE 23rd International Conference*

on Software Quality, Reliability, and Security Companion (QRS-C), pages 849–850, 2023. doi: 10.1109/QRS-C60940.2023.00107.

[10] Cheng Wang, Sunita Chandrasekaran, and Barbara Chapman. An openmp 3.1 validation testsuite. In Barbara M. Chapman, Federico Massaioli, Matthias S. Müller, and Marco Rorro, editors, *OpenMP in a Heterogeneous World*, pages 237–249, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-30961-8.

[11] Fang Yu, Shun-Ching Yang, Farn Wang, Guan-Cheng Chen, and Che-Chang Chan. Symbolic consistency checking of openmp parallel programs. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, page 139–148, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312127. doi: 10.1145/2248418.2248438. URL https://doi.org/10.1145/2248418.2248438.

A
Result tables

A.1 Static scheduling

A.1.1 Static default

Table A.1: Guided scheduling results

Guided scheduling test results				
Tests passed	Note	Compiler	Optimization flag	Loop type
10/10		g++	O0	simple
10/10		g++	O2	simple
10/10		ifx	O3	simple
10/10		nvc	default	simple
10/10		clang	default	simple
10/10		gfortran	O3	simple
10/10		g++	O3	simple
10/10		g++	default	simple
10/10		icpx	O3	simple
10/10		icpx	default	simple
10/10		gcc	O0	simple
10/10		nvc	O0	simple
10/10		icx	O2	simple
10/10		gcc	O3	simple
10/10		gfortran	O2	simple
10/10		clang	O0	simple
10/10		clang	O3	simple
10/10		clang	O1	simple
10/10		ifx	O2	simple
10/10		clang++	O2	simple
10/10		nvc	O3	simple
10/10		nvfortran	O0	simple

10/10		clang++	default	simple
10/10		icpx	O1	simple
10/10		icx	default	simple
10/10		ifx	default	simple
10/10		nvc	O2	simple
10/10		clang	O2	simple
10/10		g++	O1	simple
10/10		gcc	O2	simple
10/10		icpx	O2	simple
10/10		nvc++	default	simple
10/10		clang++	O0	simple
10/10		gcc	default	simple
10/10		clang++	O1	simple
10/10		icpx	O0	simple
10/10		nvfortran	O3	simple
10/10		nvc++	O2	simple
10/10		nvc++	O1	simple
10/10		clang++	O3	simple
10/10		ifx	O1	simple
10/10		nvfortran	O1	simple
10/10		nvc++	O0	simple
10/10		ifx	O0	simple
10/10		gcc	O1	simple
10/10		icx	O3	simple
10/10		nvc++	O3	simple
10/10		icx	O0	simple
10/10		nvfortran	default	simple
10/10		gfortran	O0	simple
10/10		icx	O1	simple
10/10		gfortran	default	simple
10/10		gfortran	O1	simple
10/10		nvfortran	O2	simple
10/10		icpc	O3	simple
10/10		icpc	O0	simple
10/10		icc	O1	simple
10/10		ifort	O3	simple
10/10		icpc	O2	simple
10/10		ifort	O1	simple
10/10		ifort	default	simple
10/10		ifort	O2	simple
10/10		ifort	O0	simple
10/10		icpc	default	simple
10/10		icc	O3	simple

10/10		icpc	O1	simple
10/10		icc	O2	simple
10/10		icc	O0	simple
10/10		g++	O0	collapse
10/10		gfortran	O0	collapse
10/10		gcc	O3	collapse
10/10		gfortran	O3	collapse
10/10		gcc	O2	collapse
10/10		icc	default	simple
10/10		g++	default	collapse
10/10		gcc	O0	collapse
10/10		g++	O2	collapse
10/10		gfortran	O1	collapse
10/10		g++	O3	collapse
10/10		gcc	default	collapse
10/10		gfortran	O2	collapse
10/10		gfortran	default	collapse
10/10		g++	O1	collapse
10/10		gcc	O1	collapse
10/10		clang	O2	collapse
10/10		clang++	default	collapse
10/10		clang	O0	collapse
10/10		clang++	O1	collapse
10/10		clang++	O2	collapse
10/10		clang++	O0	collapse
10/10		clang++	O3	collapse
10/10		clang	O1	collapse
10/10		clang	O3	collapse
10/10		clang	default	collapse
10/10		nvfortran	O3	collapse
10/10		nvc++	O1	collapse
10/10		nvc	O1	collapse
10/10		nvc	O0	collapse
10/10		nvc++	O3	collapse
10/10		nvfortran	O1	collapse
10/10		nvc++	O0	collapse
10/10		nvfortran	default	collapse
10/10		nvc++	O2	collapse
10/10		nvc	default	collapse
10/10		nvc	O3	collapse
10/10		nvfortran	O2	collapse
10/10		nvfortran	O0	collapse
10/10		nvc++	default	collapse

10/10		nvc	O2	collapse
10/10		icpx	O0	collapse
10/10		icx	O0	collapse
10/10		icpx	O3	collapse
10/10		ifx	O0	collapse
10/10		icx	default	collapse
10/10		ifx	default	collapse
10/10		icpx	O1	collapse
10/10		ifx	O2	collapse
10/10		ifx	O3	collapse
10/10		ifx	O1	collapse
10/10		icpx	O2	collapse
10/10		icx	O3	collapse
10/10		icx	O2	collapse
10/10		icpx	default	collapse
10/10		icx	O1	collapse
10/10		icc	O0	collapse
10/10		icc	O1	collapse
10/10		icc	O2	collapse
10/10		icpc	O2	collapse
10/10		ifort	O1	collapse
10/10		ifort	O2	collapse
10/10		icpc	O0	collapse
10/10		ifort	O0	collapse
10/10		icpc	O3	collapse
10/10		icc	default	collapse
10/10		ifort	O3	collapse
10/10		icc	O3	collapse
10/10		icpc	O1	collapse
10/10		ifort	default	collapse
10/10		icpc	default	collapse

A.1.2 Static with chunksize 1000

Table A.2: Guided scheduling results

Guided scheduling test results				
Tests passed	Note	Compiler	Optimization flag	Loop type
10/10		icx	default	simple
10/10		ifort	O3	simple
10/10		icpc	default	simple
10/10		icc	O2	simple

10/10		icc	O0	simple
10/10		icc	O3	simple
10/10		ifort	default	simple
10/10		gcc	O2	collapse
10/10		clang++	default	collapse
10/10		icpc	O1	simple
10/10		clang++	O3	collapse
10/10		icpc	O3	simple
10/10		ifort	O2	simple
10/10		g++	O0	simple
10/10		icx	O2	simple
10/10		clang++	O1	collapse
10/10		ifx	O1	simple
10/10		gfortran	O2	collapse
10/10		clang	O2	simple
10/10		ifx	O0	simple
10/10		nvc++	O0	collapse
10/10		ifx	default	collapse
10/10		nvc++	O3	collapse
10/10		nvfortran	O0	collapse
10/10		icx	O0	collapse
10/10		nvc++	O0	simple
10/10		g++	default	collapse
10/10		gcc	default	collapse
10/10		icx	O3	simple
10/10		nvc++	O1	collapse
10/10		clang	O0	simple
10/10		icpx	O1	collapse
10/10		clang	O1	collapse
10/10		nvfortran	O0	simple
10/10		nvc	O1	simple
10/10		icpx	O3	collapse
10/10		gcc	O3	collapse
10/10		nvc++	O2	collapse
10/10		clang++	default	simple
10/10		icpc	O0	collapse
10/10		ifort	O2	collapse
10/10		ifx	O3	simple
10/10		ifx	O2	simple
10/10		nvc	O0	simple
10/10		nvc	O2	collapse
10/10		g++	O1	simple
10/10		nvc	O3	simple

10/10		gfortran	O1	simple
10/10		gfortran	O3	simple
10/10		icx	O3	collapse
10/10		g++	default	simple
10/10		ifx	O0	collapse
10/10		ifx	O1	collapse
10/10		icc	O1	collapse
10/10		g++	O1	collapse
10/10		nvc	O0	collapse
10/10		icpc	O1	collapse
10/10		icpc	O2	simple
10/10		nvfortran	O2	simple
10/10		nvfortran	O3	collapse
10/10		icc	O2	collapse
10/10		clang	default	collapse
10/10		clang++	O0	collapse
10/10		ifx	O2	collapse
10/10		g++	O0	collapse
10/10		icpc	default	collapse
10/10		nvc	O1	collapse
10/10		ifort	O0	simple
10/10		gfortran	default	collapse
10/10		nvc++	default	collapse
10/10		icc	O1	simple
10/10		nvfortran	default	collapse
10/10		icpx	O2	collapse
10/10		icc	default	simple
10/10		g++	O3	collapse
10/10		clang	O3	collapse
10/10		icx	O2	collapse
10/10		icc	O3	collapse
10/10		nvc	O3	collapse
10/10		g++	O2	collapse
10/10		nvfortran	O1	collapse
10/10		icc	O0	collapse
10/10		clang	O0	collapse
10/10		ifx	O3	collapse
10/10		icpc	O2	collapse
10/10		gfortran	O3	collapse
10/10		icpx	O2	simple
10/10		clang	O2	collapse
10/10		icpx	O0	simple
10/10		ifort	default	collapse

10/10		nvc	default	collapse
10/10		nvc++	O1	simple
10/10		gcc	O1	simple
10/10		icpx	O1	simple
10/10		gfortran	O0	simple
10/10		icpx	default	simple
10/10		clang++	O0	simple
10/10		clang++	O3	simple
10/10		icpx	O3	simple
10/10		clang++	O1	simple
10/10		nvfortran	O1	simple
10/10		nvfortran	O3	simple
10/10		clang	default	simple
10/10		icx	O1	simple
10/10		gcc	O3	simple
10/10		gcc	O2	simple
10/10		nvc	O2	simple
10/10		nvc++	O2	simple
10/10		gfortran	default	simple
10/10		nvc++	default	simple
10/10		nvfortran	default	simple
10/10		g++	O2	simple
10/10		clang++	O2	simple
10/10		icpc	O0	simple
10/10		g++	O3	simple
10/10		clang	O1	simple
10/10		nvc	default	simple
10/10		nvc++	O3	simple
10/10		gfortran	O2	simple
10/10		icx	O0	simple
10/10		gcc	O0	collapse
10/10		gcc	default	simple
10/10		gcc	O0	simple
10/10		gfortran	O1	collapse
10/10		nvfortran	O2	collapse
10/10		gcc	O1	collapse
10/10		ifort	O1	simple
10/10		ifort	O1	collapse
10/10		icpx	default	collapse
10/10		icx	O1	collapse
10/10		clang++	O2	collapse
10/10		clang	O3	simple
10/10		icx	default	collapse

10/10		gfortran	O0	collapse
10/10		ifx	default	simple
10/10		icpx	O0	collapse
10/10		ifort	O3	collapse
10/10		icc	default	collapse
10/10		ifort	O0	collapse
10/10		icpc	O3	collapse

A.2 Dynamic scheduling

A.2.1 Dynamic with chunksize 1000

Table A.3: Guided scheduling results

Guided scheduling test results				
Tests passed	Note	Compiler	Optimization flag	Loop type
10/10		icx	O3	simple
10/10		icx	O0	simple
10/10		ifx	O3	simple
10/10		ifx	O0	simple
10/10		ifx	default	simple
10/10		icpx	O1	simple
10/10		ifx	O2	simple
10/10		icpx	O0	simple
10/10		icx	O2	simple
10/10		icpx	default	simple
10/10		icpx	O3	simple
10/10		icx	O1	simple
10/10		icx	default	simple
10/10		ifx	O1	simple
10/10		icpx	O2	simple
10/10		icpc	O2	simple
10/10		icc	O2	simple
10/10		icpc	default	simple
10/10		ifort	O2	simple
10/10		icpc	O0	simple
10/10		icc	default	simple
10/10		ifort	default	simple
10/10		icc	O3	simple
10/10		ifort	O0	simple
10/10		icc	O1	simple
10/10		ifort	O1	simple

10/10		icc	O0	simple
10/10		icpc	O1	simple
10/10		ifort	O3	simple
10/10		icpc	O3	simple
10/10		icx	O1	collapse
10/10		ifx	O0	collapse
10/10		icx	O0	collapse
10/10		icx	O2	collapse
10/10		icpx	O0	collapse
10/10		icpx	O1	collapse
10/10		ifx	O1	collapse
10/10		icpx	O2	collapse
10/10		ifx	O2	collapse
10/10		icx	O3	collapse
10/10		icpx	default	collapse
10/10		ifx	O3	collapse
10/10		icpx	O3	collapse
10/10		icx	default	collapse
10/10		ifx	default	collapse
10/10		icc	O0	collapse
10/10		ifort	O0	collapse
10/10		icc	O1	collapse
10/10		icpc	O0	collapse
10/10		icpc	O1	collapse
10/10		icc	O2	collapse
10/10		ifort	O1	collapse
10/10		icpc	O2	collapse
10/10		ifort	O2	collapse
10/10		ifort	O3	collapse
10/10		icpc	O3	collapse
10/10		icc	O3	collapse
10/10		icc	default	collapse
10/10		ifort	default	collapse
10/10		icpc	default	collapse
10/10		nvc++	O0	simple
10/10		gfortran	O3	collapse
10/10		gcc	O2	collapse
10/10		clang++	O0	simple
10/10		gfortran	O0	simple
10/10		gcc	O3	collapse
10/10		clang++	O2	collapse
10/10		gcc	default	collapse
10/10		gfortran	O1	collapse

10/10		g++	O1	collapse
10/10		g++	O3	collapse
10/10		nvfortran	O2	collapse
10/10		gfortran	O0	collapse
10/10		clang	O3	collapse
10/10		gcc	O0	collapse
10/10		gfortran	O2	collapse
10/10		g++	O2	collapse
10/10		nvc	O2	collapse
10/10		nvfortran	O3	simple
10/10		nvc	O0	simple
10/10		nvc++	O1	collapse
10/10		g++	O1	simple
10/10		nvc	O3	simple
10/10		nvfortran	O1	simple
10/10		gfortran	O3	simple
10/10		clang++	O2	simple
10/10		g++	default	simple
10/10		nvc	O1	collapse
10/10		gcc	O0	simple
10/10		nvc	O3	collapse
10/10		nvfortran	O3	collapse
10/10		g++	O0	collapse
10/10		nvc	O0	collapse
10/10		clang	O2	collapse
10/10		nvfortran	default	collapse
10/10		gcc	O1	collapse
10/10		nvc++	O3	collapse
10/10		nvc++	O2	collapse
10/10		gcc	O1	simple
10/10		nvfortran	O1	collapse
10/10		clang++	O3	collapse
10/10		clang++	O1	collapse
10/10		clang++	O0	collapse
10/10		clang	default	collapse
10/10		nvfortran	O0	collapse
10/10		nvc	default	collapse
10/10		nvc++	default	collapse
10/10		gfortran	O2	simple
10/10		nvfortran	O2	simple
10/10		nvc++	O1	simple
10/10		g++	O3	simple
10/10		nvc++	default	simple

10/10		nvc++	O0	collapse
10/10		clang++	O1	simple
10/10		clang	O2	simple
10/10		gfortran	default	simple
10/10		clang	O1	simple
10/10		nvc	default	simple
10/10		clang++	O3	simple
10/10		gcc	O3	simple
10/10		gfortran	O1	simple
10/10		clang	O3	simple
10/10		gcc	default	simple
10/10		nvc++	O3	simple
10/10		clang	default	simple
10/10		nvc++	O2	simple
10/10		nvfortran	default	simple
10/10		g++	O0	simple
10/10		gcc	O2	simple
10/10		clang	O0	simple
10/10		nvc	O1	simple
10/10		g++	default	collapse
10/10		clang	O1	collapse
10/10		clang++	default	simple
10/10		g++	O2	simple
10/10		nvfortran	O0	simple
10/10		clang	O0	collapse
10/10		gfortran	default	collapse
10/10		clang++	default	collapse

A.3 Auto scheduling

Table A.4: Guided scheduling results

Guided scheduling test results				
Tests passed	Note	Compiler	Optimization flag	Loop type
10/10	static default	gfortran	O0	simple
10/10	static default	nvc	O3	simple
10/10	static default	nvfortran	O0	simple
10/10	static default	g++	O2	simple
10/10	static default	nvc	O2	simple
10/10	static default	nvc++	default	simple
10/10	static default	gfortran	O2	simple
10/10	static default	g++	O0	simple

10/10	static default	nvc++	O1	simple
10/10	static default	g++	O1	simple
10/10	static default	gcc	O1	simple
10/10	static default	gcc	default	simple
10/10	static default	gfortran	default	simple
10/10	static default	nvfortran	default	simple
10/10	static default	gcc	O3	simple
10/10	static default	nvc	default	simple
10/10	static default	g++	default	simple
10/10	static default	gfortran	O3	simple
10/10	static default	gfortran	O1	simple
10/10	static default	g++	O3	simple
10/10	static default	nvc++	O2	simple
10/10	static default	nvc	O1	simple
10/10	static default	nvfortran	O3	simple
10/10	static default	nvc++	O0	simple
10/10	static default	nvfortran	O2	simple
10/10	static default	nvc	O0	simple
10/10	static default	nvfortran	O1	simple
10/10	static default	gcc	O0	simple
10/10	static default	gcc	O2	simple
10/10	static default	nvc++	O3	simple
10/10	dynamic	clang++	O1	simple
10/10	dynamic	clang++	O3	simple
10/10	dynamic	clang++	O2	simple
10/10	dynamic	clang	O0	simple
10/10	dynamic	clang	O3	simple
10/10	dynamic	clang	O1	simple
10/10	dynamic	clang++	default	simple
10/10	dynamic	clang	default	simple
10/10	dynamic	clang	O2	simple
10/10	dynamic	clang++	O0	simple
10/10	dynamic	icx	O0	simple
10/10	static default	gcc	O3	collapse
10/10	dynamic	icpx	O1	simple
10/10	static default	gfortran	O0	collapse
10/10	static default	g++	default	collapse
10/10	static default	gcc	O0	collapse
10/10	dynamic	icx	O1	simple
10/10	static default	gfortran	O3	collapse
10/10	static default	g++	O3	collapse
10/10	dynamic	icpx	O3	simple
10/10	static default	g++	O1	collapse

10/10	static default	g++	O2	collapse
10/10	static default	gfortran	O1	collapse
10/10	static default	g++	O0	collapse
10/10	static default	gfortran	O2	collapse
10/10	static default	gcc	default	collapse
10/10	dynamic	icpx	O0	simple
10/10	dynamic	icx	default	simple
10/10	dynamic	icx	O2	simple
10/10	static default	gcc	O1	collapse
10/10	dynamic	ifx	O2	simple
10/10	dynamic	icx	O3	simple
10/10	dynamic	ifx	O1	simple
10/10	static default	gcc	O2	collapse
10/10	static default	gfortran	default	collapse
10/10	dynamic	ifx	default	simple
10/10	dynamic	icpx	default	simple
10/10	dynamic	ifx	O3	simple
10/10	dynamic	ifort	O1	simple
10/10	dynamic	icpc	O3	simple
10/10	dynamic	ifort	O0	simple
10/10	dynamic	ifx	O0	simple
10/10	dynamic	icc	O0	simple
10/10	dynamic	icpc	default	simple
10/10	dynamic	icc	default	simple
10/10	dynamic	icc	O2	simple
10/10	dynamic	ifort	O3	simple
10/10	dynamic	icpc	O2	simple
10/10	dynamic	icpc	O0	simple
10/10	dynamic	icpx	O2	simple
10/10	dynamic	icc	O3	simple
10/10	dynamic	icpc	O1	simple
10/10	dynamic	ifort	O2	simple
10/10	dynamic	ifort	default	simple
10/10	dynamic	icc	O1	simple
10/10	static default	nvc	O2	collapse
10/10	static default	nvc++	O0	collapse
10/10	static default	nvc	O0	collapse
10/10	static default	nvfortran	O0	collapse
10/10	static default	nvc++	O1	collapse
10/10	static default	nvfortran	O2	collapse
10/10	static default	nvfortran	O1	collapse
10/10	static default	nvc	O1	collapse
10/10	static default	nvfortran	O3	collapse

10/10	static default	nvc	O3	collapse
10/10	static default	nvc++	O2	collapse
10/10	static default	nvc	default	collapse
10/10	static default	nvc++	O3	collapse
10/10	static default	nvfortran	default	collapse
10/10	static default	nvc++	default	collapse
10/10	dynamic	clang	O3	collapse
10/10	dynamic	clang	O1	collapse
10/10	dynamic	clang	default	collapse
10/10	dynamic	clang	O0	collapse
10/10	dynamic	clang++	O1	collapse
10/10	dynamic	clang++	O3	collapse
10/10	dynamic	clang++	default	collapse
10/10	dynamic	clang	O2	collapse
10/10	dynamic	clang++	O2	collapse
10/10	dynamic	clang++	O0	collapse
10/10	dynamic	icpx	O0	collapse
10/10	dynamic	icpx	O1	collapse
10/10	dynamic	ifx	O3	collapse
10/10	dynamic	ifx	O0	collapse
10/10	dynamic	icx	O3	collapse
10/10	dynamic	icx	default	collapse
10/10	dynamic	ifx	default	collapse
10/10	dynamic	ifx	O2	collapse
10/10	dynamic	icpx	O3	collapse
10/10	dynamic	ifx	O1	collapse
10/10	dynamic	icx	O0	collapse
10/10	dynamic	icpx	O2	collapse
10/10	dynamic	icx	O1	collapse
10/10	dynamic	icpx	default	collapse
10/10	dynamic	icx	O2	collapse
10/10	dynamic	ifort	O1	collapse
10/10	dynamic	ifort	O0	collapse
10/10	dynamic	icc	O1	collapse
10/10	dynamic	icpc	O1	collapse
10/10	dynamic	icc	O0	collapse
10/10	dynamic	icc	O3	collapse
10/10	dynamic	ifort	O3	collapse
10/10	dynamic	icc	O2	collapse
10/10	dynamic	icc	default	collapse
10/10	dynamic	icpc	O2	collapse
10/10	dynamic	icpc	O3	collapse
10/10	dynamic	ifort	O2	collapse

10/10	dynamic	icpc	O0	collapse
10/10	dynamic	icpc	default	collapse
10/10	dynamic	ifort	default	collapse

A.4 Guided scheduling

Table A.5: Guided scheduling results

Guided scheduling test results				
Tests passed	Note	Compiler	Optimization flag	Loop type
10/10		gcc	O1	simple
10/10		gcc	O2	simple
10/10		gcc	O0	simple
10/10		gcc	default	simple
10/10		gcc	O3	simple
0/10	modified	clang	O0	simple
0/10	modified	nvfortran	O1	simple
0/10	modified	nvfortran	O0	simple
10/10		g++	O1	simple
10/10		g++	O2	simple
0/10	modified	clang	O3	simple
0/10	modified	clang++	default	simple
10/10		g++	O3	simple
10/10		g++	O0	simple
10/10		g++	default	simple
0/10	modified	nvfortran	O2	simple
0/10	modified	clang++	O1	simple
10/10		gfortran	O2	simple
10/10		gfortran	O0	simple
10/10		gfortran	O1	simple
10/10		gfortran	default	simple
10/10		gfortran	O3	simple
0/10	modified	nvc	O2	simple
0/10	modified	clang++	O0	simple
0/10	modified	nvc	O1	simple
0/10	modified	clang++	O3	simple
0/10	modified	nvc	O0	simple
0/10	modified	nvc	O3	simple
0/10	modified	clang	O2	simple
0/10	modified	nvfortran	default	simple
0/10	modified	nvfortran	O3	simple
0/10	modified	ifx	O0	simple

0/10	modified	ifx	O2	simple
0/10	modified	ifx	O1	simple
0/10	modified	nvc++	O2	simple
0/10	modified	clang	O1	simple
0/10	modified	clang	default	simple
0/10	modified	nvc++	O1	simple
0/10	modified	nvc++	O0	simple
0/10	modified	nvc++	O3	simple
0/10	modified	clang++	O2	simple
0/10	modified	icx	O0	simple
0/10	modified	nvc	default	simple
0/10	modified	icx	O1	simple
0/10	modified	icx	O3	simple
0/10	modified	icx	O2	simple
0/10	modified	ifx	default	simple
0/10	modified	ifx	O3	simple
0/10	modified	ifort	O0	simple
0/10	modified	ifort	O2	simple
0/10	modified	ifort	O3	simple
0/10	modified	ifort	O1	simple
0/10	modified	ifort	default	simple
0/10	modified	icpx	O0	simple
0/10	modified	nvc++	default	simple
0/10	modified	icpx	O1	simple
0/10	modified	icpx	O3	simple
0/10	modified	icc	O0	simple
0/10	modified	icc	O1	simple
0/10	modified	icpx	O2	simple
10/10		gcc	O0	collapse
0/10	modified	icx	default	simple
0/10	modified	icc	default	simple
0/10	modified	icc	O3	simple
0/10	modified	icc	O2	simple
10/10		gfortran	O0	collapse
10/10		gfortran	O1	collapse
10/10		gfortran	O2	collapse
10/10		gfortran	O3	collapse
10/10		gfortran	default	collapse
10/10		gcc	O1	collapse
10/10		g++	O0	collapse
10/10		gcc	O2	collapse
10/10		gcc	default	collapse
10/10		gcc	O3	collapse

0/10	modified	icpc	O0	simple
0/10	modified	icpc	O1	simple
0/10	modified	icpx	default	simple
0/10	modified	icpc	O3	simple
0/10	modified	icpc	O2	simple
10/10		g++	O1	collapse
0/10	modified	icpc	default	simple
10/10		g++	O2	collapse
10/10		g++	default	collapse
0/10	modified	clang	O0	collapse
10/10		g++	O3	collapse
0/10	modified	clang	O1	collapse
0/10	modified	clang++	O2	collapse
0/10	modified	nvc++	O2	collapse
0/10	modified	nvc++	O0	collapse
0/10	modified	clang	default	collapse
0/10	modified	nvc++	O1	collapse
0/10	modified	nvfortran	O3	collapse
0/10	modified	ifx	O1	collapse
0/10	modified	ifx	O0	collapse
0/10	modified	nvfortran	default	collapse
0/10	modified	clang++	O0	collapse
0/10	modified	ifx	O2	collapse
0/10	modified	clang++	O1	collapse
0/10	modified	clang	O3	collapse
0/10	modified	nvfortran	O2	collapse
0/10	modified	nvc	default	collapse
0/10	modified	nvfortran	O0	collapse
0/10	modified	clang++	default	collapse
0/10	modified	nvfortran	O1	collapse
0/10	modified	icx	O2	collapse
0/10	modified	ifx	O3	collapse
0/10	modified	icx	O0	collapse
0/10	modified	icx	O1	collapse
0/10	modified	icx	O3	collapse
0/10	modified	clang	O2	collapse
0/10	modified	clang++	O3	collapse
0/10	modified	nvc	O3	collapse
0/10	modified	icpx	O0	collapse
0/10	modified	nvc++	default	collapse
0/10	modified	nvc	O1	collapse
0/10	modified	nvc	O0	collapse
0/10	modified	icpx	O2	collapse

0/10	modified	icpx	O1	collapse
0/10	modified	icx	default	collapse
0/10	modified	nvc	O2	collapse
0/10	modified	icpx	O3	collapse
0/10	modified	ifx	default	collapse
0/10	modified	ifort	O2	collapse
0/10	modified	ifort	O0	collapse
0/10	modified	ifort	default	collapse
0/10	modified	ifort	O3	collapse
0/10	modified	ifort	O1	collapse
0/10	modified	icpx	default	collapse
0/10	modified	icc	O0	collapse
0/10	modified	icc	O1	collapse
0/10	modified	icc	O3	collapse
0/10	modified	icc	O2	collapse
0/10	modified	icc	default	collapse
0/10	modified	icpc	O0	collapse
0/10	modified	icpc	O1	collapse
0/10	modified	icpc	O3	collapse
0/10	modified	icpc	O2	collapse
0/10	modified	icpc	default	collapse

B

Optimization flags

The exact flags used for standard optimization (O0, O1, O2, O3, default) are listed here, for every compiler used.

B.1 Gnu compilers (gcc, g++, gfortran)

B.1.1 -O0 (default)

-fira-region=region
-funreachable-traps

B.1.2 -O1

-fauto-inc-dec
-fbranch-count-reg
-fcombine-stack-adjustments
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fforward-propagate
-fguess-branch-probability
-fif-conversion
-fif-conversion2
-finline-functions-called-once
-fipa-modref
-fipa-profile
-fipa-pure-const
-fipa-reference
-fipa-reference-addressable

-fmerge-constants
-fmove-loop-invariants
-fmove-loop-stores
-fomit-frame-pointer
-freorder-blocks
-fshrink-wrap
-fshrink-wrap-separate
-fsplit-wide-types
-fssa-backprop
-fssa-phiopt
-ftree-bit-ccp
-ftree-ccp
-ftree-ch
-ftree-coalesce-vars
-ftree-copy-prop
-ftree-dce
-ftree-dominator-opts
-ftree-dse
-ftree-forwprop
-ftree-fre
-ftree-phiprop
-ftree-pta
-ftree-scev-cprop
-ftree-sink
-ftree-slsr
-ftree-sra
-ftree-ter
-funit-at-a-time

B.1.3 -O2

-falign-functions
-falign-jumps
-falign-labels
-falign-loops
-fcaller-saves
-fcode-hoisting
-fcrossjumping
-fcse-follow-jumps -fcse-skip-blocks
-fdelete-null-pointer-checks
-fdevirtualize -fdevirtualize-speculatively
-fexpensive-optimizations
-ffinite-loops

-fgcse -fgcse-lm
-fhoist-adjacent-loads
-finline-functions
-finline-small-functions
-findirect-inlining
-fipa-bit-cp -fipa-cp -fipa-icf
-fipa-ra -fipa-sra -fipa-vrp
-fisolate-erroneous-paths-dereference
-flra-remat
-foptimize-crc
-foptimize-sibling-calls
-foptimize-strlen
-fpartial-inlining
-fpeephole2
-freorder-blocks-algorithm=stc
-freorder-blocks-and-partition -freorder-functions
-frerun-cse-after-loop
-fschedule-insns -fschedule-insns2
-fsched-interblock -fsched-spec
-fstore-merging
-fstrict-aliasing
-fthread-jumps
-ftree-builtin-call-dce
-ftree-loop-vectorize
-ftree-pre
-ftree-slp-vectorize
-ftree-switch-conversion -ftree-tail-merge
-ftree-vrp
-fvect-cost-model=very-cheap

B.1.4 -O3

-fgcse-after-reload
-fipa-cp-clone
-floop-interchange
-floop-unroll-and-jam
-fpeel-loops
-fpredictive-commoning
-fsplit-loops
-fsplit-paths
-ftree-loop-distribution
-ftree-partial-pre
-funswitch-loops

-fvect-cost-model=dynamic
-fversion-loops-for-strides

B.2 Clang compilers (clang, clang++, flang

LLVM optimization flags have been retrieved using

```
llvm-as < /dev/null | opt -O1 -disable-output -debug-pass=Arguments
```

as suggested here

B.2.1 -O0 (default)

-tti
-verify
-ee-instrument
-targetlibinfo
-assumption-cache-tracker
-profile-summary-info
-forceattrs
-basiccg
-always-inline
-barrier
-verify

B.2.2 -O1

–tti
-tbaa
-scoped-noalias
-assumption-cache-tracker
-targetlibinfo
-verify
-ee-instrument
-simplifycfg
-domtree
-sroa
-early-cse
-lower-expect
-profile-summary-info
-forceattrs
-inferattrs
-ipsccp
-called-value-propagation

-attributor
-globalopt
-mem2reg
-deadargelim
-basicaa
-loops
-lazy-branch-prob
-lazy-block-freq
-opt-remark-emitter
-instcombine
-basiccg
-globals-aa
-prune-eh
-always-inline
-functionattrs
-memoryssa
-early-cse-memssa
-libcalls-shrinkwrap
-pgo-memop-opt
-reassociate
-loop-simplify
-lcssa-verification
-scalar-evolution
-loop-rotate
-licm
-loop-unswitch
-indvars
-loop-idiom
-loop-deletion
-loop-unroll
-phi-values
-memdep
-memcpyopt
-demanded-bits
-bdce
-postdomtree
-adce
-barrier
-rpo-functionattrs
-globaldce
-float2int
-lower-constant-intrinsics
-loop-accesses

-loop-distribute
-loop-vectorize
-loop-load-elim
-transform-warning
-alignment-from-assumptions
-strip-dead-prototypes
-loop-sink
-instsimplify
-div-rem-pairs

B.2.3 -O2

–tti
-tbaa
-scoped-noalias
-assumption-cache-tracker
-targetlibinfo
-verify
-ee-instrument
-simplifycfg
-domtree
-sroa
-early-cse
-lower-expect
-profile-summary-info
-forceattrs
-inferattrs
-ipsccp
-called-value-propagation
-attributor
-globalopt
-mem2reg
-deadargelim
-basicaa
-loops
-lazy-branch-prob
-lazy-block-freq
-opt-remark-emitter
-instcombine
-basiccg
-globals-aa
-prune-eh
-inline

-functionattrs
-memoryssa
-early-cse-memssa
-speculative-execution
-lazy-value-info
-jump-threading
-correlated-propagation
-libcalls-shrinkwrap
-pgo-memop-opt
-tailcallelim
-reassociate
-loop-simplify
-lcssa-verification
-scalar-evolution
-loop-rotate
-licm
-loop-unswitch
-indvars
-loop-idiom
-loop-deletion
-loop-unroll
-mldst-motion
-phi-values
-memdep
-gvn
-memcpyopt
-demanded-bits
-bdce
-dse
-postdomtree
-adce
-barrier
-elim-avail-extern
-rpo-functionattrs
-globaldce
-float2int
-lower-constant-intrinsics
-loop-accesses
-loop-distribute
-loop-vectorize
-loop-load-elim
-slp-vectorizer
-transform-warning

-alignment-from-assumptions
-strip-dead-prototypes
-constmerge
-loop-sink
-instsimplify
-div-rem-pairs
-block-freq-targetlibinfo

B.2.4 -O3

–tti
-tbaa
-scoped-noalias
-assumption-cache-tracker
-targetlibinfo
-verify
-ee-instrument
-simplifycfg
-domtree
-sroa
-early-cse
-lower-expect
-profile-summary-info
-forceattrs
-inferattrs
-callsite-splitting
-ipsccp
-called-value-propagation
-attributor
-globalopt
-mem2reg
-deadargelim
-basicaa
-loops
-lazy-branch-prob
-lazy-block-freq
-opt-remark-emitter
-instcombine
-basiccg
-globals-aa
-prune-eh
-inline
-functionattrs

-argpromotion
-memoryssa
-early-cse-memssa
-speculative-execution
-lazy-value-info
-jump-threading
-correlated-propagation
-aggressive-instcombine
-libcalls-shrinkwrap
-pgo-memop-opt
-tailcallelim
-reassociate
-loop-simplify
-lcssa-verification
-scalar-evolution
-loop-rotate
-licm
-loop-unswitch
-indvars
-loop-idiom
-loop-deletion
-loop-unroll
-mldst-motion
-phi-values
-memdep
-gvn
-memcpyopt
-demanded-bits
-bdce
-dse
-postdomtree
-adce
-barrier
-elim-avail-extern
-rpo-functionattrs
-globaldce
-float2int
-lower-constant-intrinsics
-loop-accesses
-loop-distribute
-loop-vectorize
-loop-load-elim
-slp-vectorizer

-transform-warning
-alignment-from-assumptions
-strip-dead-prototypes
-constmerge
-loop-sink
-instsimplify
-div-rem-pairs
-block-freq

B.3 NVIDIA compilers (nvc, nvc++, nvfortran)

No list of specific optimization flags has been found in the documentation [2] for the NVIDIA compiler set. Therefore only abstract descriptions can be provided.

B.3.1 Default

When no level is specified, level two global optimizations are performed, including traditional scalar optimizations, induction recognition, and loop invariant motion. No SIMD vectorization is enabled.

B.3.2 -O0

Level zero specifies no optimization. A basic block is generated for each language statement. At this level, the compiler generates a basic block for each statement.
Performance will almost always be slowest using this optimization level. This level is useful for the initial execution of a program. It is also useful for debugging, since there is a direct correlation between the program text and the code generated. To enable debugging, include -g on your compile line.

B.3.3 -O1

Level one specifies local optimization. Scheduling of basic blocks is performed. Register allocation is performed.
Local optimization is a good choice when the code is very irregular, such as code that contains many short statements containing IF statements and does not contain loops (DO or DO WHILE statements). Although this case rarely occurs, for certain types of code, this optimization level may perform better than level-two (-O2).

B.3.4 -O2

Level two specifies global optimization. This level performs all level-one local optimizations as well as level two globl optimization described in -O. In addition, more advanced optimizations such as SIMD code generation, cache alignment, and partial redundancy eliminitaion

[2] https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/#cmd-line-opts

are enabled.

B.3.5 -O3

Level three specifies aggressive global optimization. This level performs all level-one and level-two optimizations and enables more aggressive hoisting and scalar replacement optimizations that may or may not be profitable.

B.4 Intel classic compilers

No list of specific optimization flags has been found in the documentation [3] for the Intel OneAPI compiler set. Therefore only abstract descriptions can be provided.

B.4.1 -O0

No optimization. Used during the early stages of application development and debugging.

B.4.2 -O1

Optimize for size. Omits optimizations that tend to increase object size. Creates the smallest optimized code in most cases. May be useful in large server/database applications where memory paging due to larger code size is an issue.

B.4.3 -O2

Maximize speed. Default setting. Enables many optimizations, including vectorization and intra-file interprocedural optimizations. Creates faster code than /O1 (-O1) in most cases.

B.4.4 -O3

Enables /O2 (-O2) optimizations plus more aggressive loop and memory-access optimizations, such as scalar replacement, loop unrolling, loop blocking to allow more efficient use of cache and additional data prefetching. The /O3 (-O3) option is particularly recommended for applications that have loops that do many floating-point calculations or process large data sets. These aggressive optimizations may occasionally slow down other types of applications compared to /O2 (-O2).

B.5 Intel OneAPI compilers

No list of specific optimization flags has been found in the documentation [4] for the Intel OneAPI compiler set. Therefore only abstract descriptions can be provided.

[3] https://cdrdv2-public.intel.com/671303/quick-reference-guide-intel-compilers-v19-1-final-.pdf
[4] https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2023-0/o-001.html

B.5.1 -O0

Disables all optimizations.
This option may set other options. This is determined by the compiler, depending on which operating system and architecture you are using. The options that are set may change from release to release.

B.5.2 -O1

Enables optimizations for speed and disables some optimizations that increase code size and affect speed. To limit code size, this option:
Enables global optimization; this includes data-flow analysis, code motion, strength reduction and test replacement, split-lifetime analysis, and instruction scheduling.
Disables inlining of some intrinsics.
This option may set other options. This is determined by the compiler, depending on which operating system and architecture you are using. The options that are set may change from release to release.
The O1 option may improve performance for applications with very large code size, many branches, and execution time not dominated by code within loops.

B.5.3 -O2

Enables optimizations for speed. This is the generally recommended optimization level. Vectorization is enabled at O2 and higher levels.
This option also enables:

- Inlining of intrinsics
- Intra-file interprocedural optimization, which includes:
- inlining
- constant propagation
- forward substitution
- routine attribute propagation
- variable address-taken analysis
- dead static function elimination
- removal of unreferenced variables
- The following capabilities for performance gain:
- constant propagation
- copy propagation
- dead-code elimination

- global register allocation
- global instruction scheduling and control speculation
- loop unrolling
- optimized code selection
- partial redundancy elimination
- strength reduction/induction variable simplification
- variable renaming
- exception handling optimizations
- tail recursions
- peephole optimizations
- structure assignment lowering and optimizations
- dead store elimination

This option may set other options, especially options that optimize for code speed. This is determined by the compiler, depending on which operating system and architecture you are using. The options that are set may change from release to release.
This content does not apply to SYCL. On Linux systems, the -debug inline-debug-info option will be enabled by default if you compile with optimizations (option -O2 or higher) and debugging is enabled (option -g).
Many routines in the shared libraries are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

B.5.4 -O3

Performs O2 optimizations and enables more aggressive loop transformations such as Fusion, Block-Unroll-and-Jam, and collapsing IF statements.
This option may set other options. This is determined by the compiler, depending on which operating system and architecture you are using. The options that are set may change from release to release.
The O3 optimizations may not cause higher performance unless loop and memory access transformations take place. The optimizations may slow down code in some cases compared to O2 optimizations.
The O3 option is recommended for applications that have loops that heavily use floating-point calculations and process large data sets.
Many routines in the shared libraries are more highly optimized for Intel® microprocessors than for non-Intel microprocessors.

Universität Basel

Philosophisch-Naturwissenschaftliche Fakultät

Erklärung zur wissenschaftlichen Redlichkeit und Veröffentlichung der Arbeit (beinhaltet Erklärung zu Plagiat und Betrug)

Titel der Arbeit: Validation and Verification of the OpenMP Standard functionality with Focus on Scheduling Clauses

Name Beurteiler*in: Florina Ciorba

Name Student*in: Sascha Maibach

Matrikelnummer: 19-932-441

Ich bezeuge mit meiner Unterschrift, dass ich meine Arbeit selbständig ohne fremde Hilfe verfasst habe und meine Angaben über die bei der Abfassung meiner Arbeit benützten Quellen in jeder Hinsicht der Wahrheit entsprechen und vollständig sind. Alle Quellen, die wörtlich oder sinngemäss übernommen wurden, habe ich als solche gekennzeichnet.
Des Weiteren versichere ich, sämtliche Textpassagen, die unter Zuhilfenahme KI-gestützter Programme verfasst wurden, entsprechend gekennzeichnet sowie mit einem Hinweis auf das verwendete KI-gestützte Programm versehen zu haben.
Eine Überprüfung der Arbeit auf Plagiate und KI-gestützte Programme – unter Einsatz entsprechender Software – darf vorgenommen werden. Ich habe zur Kenntnis genommen, dass unlauteres Verhalten zu einer Bewertung der betroffenen Arbeit mit einer Note 1 oder mit «nicht bestanden» bzw. «fail» oder zum Ausschluss vom Studium führen kann.

Ort, Datum: Gelterkinden, 20.1.2025 Student*in: S. Maibach

Wird diese Arbeit oder Teile davon veröffentlicht?

☐ Nein

☑ Ja. Mit meiner Unterschrift bestätige ich, dass ich mit einer Veröffentlichung der Arbeit (print/digital) in der Bibliothek, auf der Forschungsdatenbank der Universität Basel und/oder auf dem Dokumentenserver des Departements / des Fachbereichs einverstanden bin. Ebenso bin ich mit dem bibliographischen Nachweis im Katalog SLSP (Swiss Library Service Platform) einverstanden. (nicht Zutreffendes streichen)

Veröffentlichung ab: Februar 2025

Ort, Datum: Gelterkinden, 20.1.2025 Student*in: S. Maibach

Ort, Datum: _____ Beurteiler*in: _____

Diese Erklärung ist in die Bachelor-, resp. Masterarbeit einzufügen.