FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

# Automatic Performance Measurements and Data Evaluation for Highly Parallel Codes

Oleg Kravchuk

Master's Thesis

# Automatic Performance Measurements and Data Evaluation for Highly Parallel Codes

Oleg Kravchuk

Master's Thesis

| | |
|---|---|
| Aufgabensteller: | Dr.-Ing. habil. Harald Köstler |
| Betreuer: | M. Sc. Sebastian Kuckuk |
| Bearbeitungszeitraum: | 1.8.2014 – 2.2.2015 |

# List of Figures

# List of Tables

# Listings

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

The purpose of this work is to develop a toolchain for automated performance measurement and measured data analysis of highly parallel programs, e.g. MPI- and/or OpenMP-based. There exist many different ways of performance measurement which are used in high performance computing. In this work, however, we refer to the most low-level ways to measure time, which are available on modern hardware and operating systems used for HPC.

This work is written as a part of ExaStencils [2] project and, thus, is coupled with Exastencils metacompiler and its generation flow. Thus, Exastencils metacompiler is the tool which provides the code to measure.

This work consists of 3 main parts:
- Developing appropriate measurement tool
- Creating toolchain for automated compilation and evaluation of given code.
- Coupling between code generation and code profiling with genetic algorithm.

## 1.2 Overview

### Performance measurement approaches

In this chapter we will touch upon performance measurement approaches. The main form of performance measurement is profiling.

Profiling is a form of dynamic program analysis. This involves measurement of time and/or memory used for arbitrary program or its part. Profiling is focused on dynamic execution and, thus, reveals interaction between application and the host machine. Data produced by profiler is called "profile" and is used for deeper understanding of program behaviour. Profiling is necessary for understanding the efficiency of the program itself and efficiency of hardware in scope of algorithm used in profiled application.

There are several ways to profile an application:

- Sampling
- Source code instrumentalization
- Binary instrumentalization
- Use of hypervisor

### Sampling

Sampling is the simplest way to profile an application. This is done by external application, called Sampling Profiler, which interrupts processor (CPU) periodically (at sampling points) and takes a snapshot of current call stack. There are inclusive and exclusive samples. Inclusive tracks call stack entries including its children, whereas exclusive does not take into account the children calls.

Sampling allows to gather call graph and some information about execution flow. Due to stochastic nature of profiles generated, the main drawback of this method is inaccuracy if applied to short in time programs. Overhead of sampling depends on sampling frequency - the more samples you do, the more impact it causes on performance.

Pros:

- Easy to use
- Does not require modification of binary

Cons:

- Inaccurate for short in time programs
- Requires long-term runs to achieve statistically significant values

Examples:

- VerySleepy
- Gprof
- Perf
- TAU

Conclusion:

Sampling is good for long-term running applications which are doing the same work during measurement period - games, renderers and so on. But for HPC and other applications which run in a limited period of time sampling is not really suitable due to relatively low precision.

### Source code instrumentalization

Source code instrumentalization (SCI) in scope of this work refers to modification of source code of the program by adding special markers and/or calls to external functions in order to collect profile data. The simplest SCI is about adding special markers at desired locations and recording program state or time at the moment when execution reaches this marker. It allows to specify what kind of information we want to get as profile. Thus, profile can be anything like memory usage, time stamp, current cache state and so on.

Pros:

- Allows to have profiling regions
- Detailed information

Cons:

- Requires source code modification and, thus, recompilation

Examples:

- TAU
- Likwid
- Manually defined functions, e.g. printf(...) or Trace.WriteLine("...")

Conclusion:
In comparison to sampling, overhead caused by SCI is rather constant but low and depends only on how often instrumentalized function is called.

### Binary instrumentalization

Binary instrumentalization (BI) modifies application at runtime (runtime injection), right before execution (runtime instrumentalization) or in prior to execution (static instrumentalization). The runtime injection is the lightweight version of runtime instrumentalization.

Pros:

- Detailed information
- Generally causes less overhead than sampling

Cons:

- Generally does not allow to have profiling regions

Examples:

- TAU
- Scalasca (allows exclude list)
- Pin
- QEMU
- ATOM
- DinInst

Conclusion:
Binary instrumentalization requires no direct modification of binary by user (this is done automatically) but often creates much of overhead and therefore can change measured data significantly. Yet this impact depends on what data is collected and its level of detail. As for this work, BI has high performance impact, and, generally, does not allow to have just the regions of interest to be profiled.

#### Hypervisor

Hypervisor acts like virtual environment where application is executed and, thus, shows all data and instruction flows created by the program. However, hypervisor is out of scope of this research since it causes performance degradation. That is because the code is executed on a virtual machine instead of real hardware.

## Requirements for performance measurement tool

As a part of a greater project, there are several requirements for measurement tool:

- Profiling regions
- Minimum possible overhead
- Has to be portable
- Possibility to generate profile in suitable format

#### Profiling regions

Profiling regions are the regions of interest. We specify them by imposing specific start/stop markers and measure time within these regions. Also, each profiling region must be within one scope.

#### Minimum possible overhead

This is necessary due to the highly computationally intensive nature of the code. If a profiler will interfere too much with the inspected program this will lead to inaccurate results. This also includes calls to external libraries which will decrease performance.

#### Portable

It must be possible to use this tool on different platforms and on different operation systems.

#### Possibility to generate profile in suitable format

This is necessary as the measured profile will be used for further processing and there should be a way to aggregate and save measured data for further post-processing.

#### Conclusion

To fulfill these requirements the source code instrumentalization was selected, as recompilation is not the issue due to generated nature of the inspected code. This allows to directly insert region markers at desired positions of the code, collect only data we want, and write measured data to storage when it is necessary in a format that we set. In order to be portable, the measurement is done by a special class "Stopwatch" which incorporates calls to a particular time measurement function, available on target platform. For batch- and post- processing the special scripting infrastructure will be presented, capable of performing all stages of performance evaluation including generation, compilation and execution of inspected program.

## Scripting infrastructure

In this thesis Python [3] scripting language is used to write scripts for execution management and data post processing.

Python is a high-level scripting programming language with a huge standard library and support community. It is a multiparadigm programming language: object-oriented programming and structured programming are fully supported, and there are a number of language features which support functional programming and metaprogramming. Python has extensive math library and is frequently used as a scientific scripting language, for instance: FEniCS [4], SciPy [5].

# Chapter 2

# Stopwatch

*"Time is the simplest thing"*

— Clifford D. Simak

## 2.1 Overview

One of the most important parts of performance measurement system is the interface - how the system should be called within the measured application.

These are the following criteria for the interface:

- Regioning is the possibility to have start/stop markers for performance[1] measurement system.
- Output is the possibility to print measured data to console or to file for future processing

There are several use-cases for region to be defined within the code:

- Simple region (meaning no inclusion of other timers within the region)
- Nested (or inclusive) region.
- Nested region with a loop
- Nested region with recurrence

### Simple region

Simple region is defined by imposing Start() and Stop() markers at desired location in the same execution scope:

Listing 2.1: Simple profiling region

```
...
TIMER_START
// RegionOfInterest
foo();
...
TIMER_STOP
...
```

[1]Performance, in context of this thesis, is the time, spent by program within some region. Time between Start() and Stop() markers are treated as time (performance) of a region. The less time is spent, the better the performance is.

## Nested region

Nested (or inclusive) region means that one timer is called within region of another (outer) timer:

Listing 2.2: Inclusive profiling region

```
...
TIMER_START
foo();
TIMER_START
bar();
TIMER_STOP
TIMER_STOP
...
```

There are three ways of understanding this example. On one hand, these are two different timers, one being called within another. On the other hand, it is the same timer, which can be interpreted as a necessity to create an additional instance of timer. Another option would be to treat inner call as continuation of outer call (no additional instance is created) and, therefore, only outer call is being measured.

## Nested region with loop

Use-case for loop is the extension of a previous example:

Listing 2.3: Inclusive profiling region with loop

```
...
TIMER_START
for(uint i = 0; i<goal; i++)
{
        ...
        TIMER_START
        foo();
        TIMER_STOP
        ...
}
TIMER_STOP
...
```

The only difference is that inner timer is called multiple times.

## Nested region with recurrence

The picture changes if there is a recurrent call:

- Simple case:

Listing 2.4: Simple recurrence profiling region

```
...
void foo(...)
{
        ...
        TIMER_START
        //do something
        TIMER_STOP
        foo(...);
        ...
}
```

- Inclusive call:

Listing 2.5: Inclusive recurrence profiling region

```
...
void foo(...)
{
        ...
        TIMER_START
        //do something
        foo(...);
        TIMER_STOP
        ...
}
```

Simple case will result in sequential call of one timer. The amount of pairs of Start()-Stop() calls is equal to the recursion depth of foo(...) function.

In case of inclusive call, creating new instance of timer for each inclusion can be very expensive. Hence, the optimal way to treat this would be the following: every inner timer call is treated as continuation of the most outer call of the same timer, and the final measured value will be the one, measured by the most outer call.

If call stack tracking is necessary, then a new instance of timer should be created for every inclusion. A more closer look at this will be presented in Implementation section.

## Profile output

After the measurements are done the program should output the data into console or to file. Printing should occur after all calculations are finished, generally right before exit.

Listing 2.6: Printing measured data

```
void main()
{
        ...
        PRINT_TIMERS
        exit();
}
```

## Storage for timers

These timers should have some global storage and they should be created on application start and destroyed on application exit.

The management of this storage is up to the end user. Observe a simple example of the namespace as storage:

Listing 2.7: Example storage for timers

```
//.h
namespace Timers
{
        extern Timer _timer1;
        extern Timer _timer2;
}
//.cpp
```

```
namespace Timers
{
        Timer _timer1("Timer1");
        Timer _timer2("Timer2");
}
```

Example usage:

Listing 2.8: Example usage of timers

```
...
Timers::_timer1.Start();
//RegionOfInterest
foo();
...
Timers::_timer1.Stop();
...
```

This set of features is sufficient in scope of this thesis. They allow to have measurement regions and to deal with recurrent and nested calls. The actual implementation of such a timer will be further discussed in the next section.

## 2.2 Implementation

### Time measurement

The class declaration is simple enough:

Listing 2.9: Stopwatch interface

```
class Stopwatch
{
public:
        Stopwatch(void);
        Stopwatch(const std::string& name);
        ~Stopwatch(void);

        void Start();
        void Stop();
        void Reset();

        double getTotalTimeInMilliSec();
        ...
};
```

The class is capable of tracking multiple calls of the same timer in order to deal with recurrence calls. Main idea is that the timer tracks every Start() call, incrementing entry counter:

Listing 2.10: Entry counter start

```
void Start()
{
                if (!_entries)
                {
                                _time_start = TIMER_NOW;
                                _mcs_local = TIMER_ZERO;

                                ++_total_entries;
                }

                ++_entries;
}
```

Stop() method behaves similarly:

Listing 2.11: Entry counter stop

```
void Stop()
{
        if ( _entries == 0 )
        {
                _name = "Invalid"; //Check your code, you stop before start.
        }
        if (_entries == 1) // if you don't properly stop all calls, _ns_local will be 0.
        {
                _mcs_local = TIMER_GET_TIME_STORE(TIMER_NOW -_time_start);
                _mcs_global += _mcs_local;
        }
        --_entries;
}
```

If at some point *_entries* exceeds 1, then we know that there is an inclusive call of a recurrent function or a nested call ( see listings 2.5, 2.2 ). This helps to avoid unnecessary timer creation and measurement continues producing a more accurate result of inclusive measurement within recurrent call in contrast to recreating the timer.

On the other hand, this leads to having only the most outer call measured ( as if there is no recurrence):

Listing 2.12: Inclusive recurrent call unwrapped

```
void foo(...)
{
        ...
        TIMER_START//START_TIME is saved
        //do something
        call foo(...)
        {
                ...
                TIMER_START //increases counter
                //do something
                call foo(...)
                        ....
                TIMER_STOP//decreases counter
                ...
        }
        TIMER_STOP//measured time is START_TIME - TIME_NOW
        ...
}
```

Yet this allows to measure inclusive time at a relatively low cost and generate reliable profile for application.

## Call stack tracking

Another interesting challenge is to build a call graph produced by timers. In order to do this there should be a way to trace every Start() and Start() call of every timer. The solution to this task in my work is presented here.

Firstly, there should be some entity that emulates the timer behaviour. Also, every entry can have multiple children (child calls). We will use it to build a graph. Secondly, there should be some sort of tracker that will actually build a graph.

Building the call graph (or call stack) will decrease performance, but will give the idea of how much time is spent on every level of a call tree. Also this allows us to compare the behaviour of application, executed on different platforms, hardware and operating systems.

Hence we conclude that following classes are necessary:

- Call entity
- Call tracker

### Call entity

Call entity is the granular entity that is first created on $Start()$ call. New entity should be created on every call unless it has been created before. A timer can have multiple children and one parent.

The interface of such an entity looks like this:

Listing 2.13: CallEntity interface

```
class CallEntity
{
friend class CallTracker;
public:
        CallEntity(CallEntity* parent, Stopwatch* bt);
        ~CallEntity();

        void Start();
        void Stop();

        bool CheckHasTimer(Stopwatch* bt, CallEntity** ent);

        void AddChild(CallEntity* child);
        ...
};
```

### Call tracker

Call tracker is the one that tracks every call of every Start() and Stop() call of every timer within the program:

Listing 2.14: CallTracker interface

```
class CallTracker
{
        friend class Stopwatch;
private:
        static void StartTimer( Stopwatch* timer );
        static void StopTimer( Stopwatch* timer );
public:
        static void PrintCallStack();
        static void ClearCallStack();
private:
        static CallEntity** GetHead();
        static void Print( int displacement, CallEntity* root );
        static void Clear( CallEntity* root );
...
};
```

In order to use this tracker there should be modification imposed to 2.10:

Listing 2.15: Call stack start

```
void Start()
{
...
        CallTracker::StartTimer(this);
}
```

And modifications to 2.11:

Listing 2.16: Call stack stop

```
void Stop()
{
....
        CallTracker::StopTimer(this);
}
```

Call tracker will build a graph with entities as nodes. Tracker always has current "head" - entity that represents the last timer, for which $Start()$ was called. If $Stop()$ is called for this timer, entity will stop measurement as well and "head" will be changed to entity's parent.

After main execution is done and all timers are stopped, this graph can be saved to file or printed to console.

Call graph can be used to investigate execution flow of application in details and, thus, can be an additional option for application profiling.

## Macro defines

Up to this point Stopwatch class had an interface and implementation but there is nothing yet said about the actual time measurement functions. This produces one more layer of abstraction, where in Stopwatch all data and functions, related to time measurement, are hidden behind macro definitions:

- TIMER_ZERO for zero in current time format
- TIMER_NOW for 'now' in current time format
- TIMER_TIME_STORE_TYPE for current time format
- TIMER_TIME_TIMEPOINT_TYPE for timepoint in current time format
- TIMER_GET_TIME_STORE(dif) for cast from time interval to current time format
- TIMER_GET_TIME_MILI(dif) for cast from time interval to milliseconds

Such amount of defines is used since c++11 <chrono> requires each of these, and it is the maximum amount of definitions across all tested time measurement functions.

This allows to write timer implementation in the following way:

Listing 2.17: Example usage of macro defines

```
...
TIMER_TIME_TIMEPOINT_TYPE _time_start = TIMER_NOW;
TIMER_TIME_STORE_TYPE _mcs_global = TIMER_ZERO;
...
```

Examples of defines for different time measurement functions are presented in the next section.

## 2.3 Macro defines examples

Since the specific timer for Stopwatch class is defined by macroses, below are listed different definitions for different time measurement functions:

Listing 2.18: Macro defines for std::chrono

```
#       include <chrono>
using namespace std::chrono;
using std::chrono::nanoseconds;
#       define TIMER_ZERO nanoseconds::zero()
#       define TIMER_NOW high_resolution_clock::now()
#       define TIMER_TIME_STORE_TYPE nanoseconds
#       define TIMER_TIME_TIMEPOINT_TYPE high_resolution_clock::time_point
#       define TIMER_GET_TIME_STORE(dif) duration_cast<nanoseconds>(dif)
#       define TIMER_GET_TIME_MILI(dif) (duration_cast<nanoseconds>(dif).count())*1e-6
```

Listing 2.19: Macro defines for QPC

```
#       include <windows.h>
#       define TIMER_TIME_STORE_TYPE long long
#       define TIMER_TIME_TIMEPOINT_TYPE long long

inline TIMER_TIME_TIMEPOINT_TYPE milliseconds_now()
{
        LARGE_INTEGER now;
        QueryPerformanceCounter(&now);
        return (now.QuadPart);
}

inline double to_milisec(TIMER_TIME_STORE_TYPE val)
{
        static LARGE_INTEGER s_frequency;
        static BOOL s_use_qpc = QueryPerformanceFrequency(&s_frequency);
        return val/(s_frequency.QuadPart/1000.0);
}

#       define TIMER_ZERO 0
#       define TIMER_NOW milliseconds_now()

#       define TIMER_GET_TIME_STORE(dif) dif
#       define TIMER_GET_TIME_MILI(dif) to_milisec(dif)
```

Listing 2.20: Macro defines for Windows clock()

```
#   include <time.h>
#       include <sys/types.h>
#       define TIMER_ZERO 0.0
inline double time_now() //millisec.
{
        clock_t t = clock();
        return (((double)t)/(CLOCKS_PER_SEC* 1e-3));
};

#       define TIMER_NOW time_now()
#       define TIMER_TIME_STORE_TYPE double //assumption - primary time in milliseconds
#       define TIMER_TIME_TIMEPOINT_TYPE double
#       define TIMER_GET_TIME_STORE(dif) dif
#       define TIMER_GET_TIME_MILI(dif) dif
```

Listing 2.21: Macro defines for Unix gettimeofday()

```
#       include <sys/time.h>
#       include <sys/types.h>
inline double time_now() //millisecs. unix
{
        timeval timePoint;
        gettimeofday(&timePoint, NULL);
        return (double)(timePoint.tv_sec) * 1e3 + (double)(timePoint.tv_usec) * 1e-3;
}
#       define TIMER_NOW time_now()
#       define TIMER_ZERO 0.0
#       define TIMER_TIME_STORE_TYPE double //assumption - primary time in milliseconds
#       define TIMER_TIME_TIMEPOINT_TYPE double
#       define TIMER_GET_TIME_STORE(dif) dif
#       define TIMER_GET_TIME_MILI(dif) dif
```

Listing 2.22: Macro defines for MPI_Wtime()

```
#       include <mpi.h>
#       define TIMER_ZERO 0.0
#       define TIMER_NOW MPI_Wtime()
#       define TIMER_TIME_STORE_TYPE double //assumption - primary time in milliseconds
#       define TIMER_TIME_TIMEPOINT_TYPE double
#       define TIMER_GET_TIME_STORE(dif) dif
#       define TIMER_GET_TIME_MILI(dif) dif*1e3
```

Listing 2.23: Macro defines for Windows RDTSC

```
#       define FREQ YOUR_CPU_QREFUENCY_IN_HZ
#       define TIMER_TIME_STORE_TYPE long long
#       define TIMER_TIME_TIMEPOINT_TYPE long long

#       pragma intrinsic(__rdtsc)
inline TIMER_TIME_STORE_TYPE now()
{
        return __rdtsc();
}
#       define TIMER_ZERO 0
#       define TIMER_NOW now()
#       define TIMER_GET_TIME_STORE(dif) dif
#       define TIMER_GET_TIME_MILI(dif) dif/(FREQ/1000.0)
}
```

Listing 2.24: Macro defines for Linux RDTSC

```
#       define FREQ YOUR_CPU_QREFUENCY_IN_HZ
#       define TIMER_TIME_STORE_TYPE long long
#       define TIMER_TIME_TIMEPOINT_TYPE long long

#       include <stdint.h>
extern __inline__ TIMER_TIME_STORE_TYPE rdtsc()
{
        TIMER_TIME_STORE_TYPE x;
        __asm__ volatile ("rdtsc\n\tshl␣$32,%%rdx\n\tor␣%%rdx,␣%%rax" : "=a" (x) : : "rdx");
        return x;
}
#       define TIMER_ZERO 0
#       define TIMER_NOW now()
#       define TIMER_GET_TIME_STORE(dif) dif
#       define TIMER_GET_TIME_MILI(dif) dif/(FREQ/1000.0)
}
```

## 2.4 Example usage

After Stopwatch class was implemented, we present example usage. This can be treated as test case for Stopwatch class, and can help investigate its actual performance.

For this test the following configuration was used:

- OS: Windows 7 x64
- Compiler: VS2012
- CPU: Intel® Core™ i7-3610QM 2.3GHz
- Memory: 1600 MHz DDR3 in dual channel

Since we are working on a test case simple function was written - recurrence with some calculation inside and two timers:

Listing 2.25: Test recurrence function with inclusive timer call

```
int fib(int x) {
        if (x == 1 || x == 0)
        {
                return 1;
        }
        else
        {
                ProfilingTimers::_timer1.Start();
                int tt = heavy(40);// some heavy function
                ProfilingTimers::_timer1.Stop();
                ProfilingTimers::_timer2.Start();
                tt += fib(x-1) + fib(x-2);
                ProfilingTimers::_timer2.Stop();
                return tt;
        }
}
...
void TestProfiling()
{
        int result = fib(10);
        std::cout<< std::endl;
        std::cout<< ProfilingTimers::_timer1.GetName()<<":"<< std::endl;
        std::cout<< ProfilingTimers::_timer1.getMeanTimeInMilliSec() << std::endl;
        std::cout<< std::endl;
        std::cout<< ProfilingTimers::_timer2.GetName()<<":"<< std::endl;
        std::cout<< ProfilingTimers::_timer2.getMeanTimeInMilliSec() << std::endl;

        CallTracker::PrintCallStack();

}
```

This set up allows to investigate the behaviour of both inclusive and non-inclusive timers. Profiled output for this example looks like following[2]:

Listing 2.26: Output of inclusive recurrence with call stack profiling 1

```
Timer1: 635.38 ms

Timer2: 55352 ms

Call stack:
 Timer1;635.37; ms
 Timer2;55352.000000; ms
```

[2]One can be curious why Timer1 has different values along the stack. The reason is that recurrent function spawns two child functions every time, so same CallEntity is called more than once at one recurrence level. In other worlds, it is measuring parallel recursion as if it is simple recurrence.

```
  Timer1;1281.000000; ms
  Timer2;54069.000000; ms
....
         Timer1;5066.000000; ms
         Timer2;633.000000; ms
          Timer1;632.000000; ms
          Timer2;0.000000; ms
```

In order to investigate the cost of call stack tracking we need to remove tracking and compare measured results. Without call stack tracking measured times are different:

Listing 2.27: Output of inclusive recurrence without call stack profiling 1

```
Timer1:
634.227 ms

Timer2:
55125 ms
```

Relative speedup is:

- For timer_1 is 0.018
- For timer_2 is 0.041

Here it looks like call stack tracking doesn't influence measured data much. But picture will change if the heavy() function will be removed, since most of time will be spent on call stack management.

With the call stack tracking:

Listing 2.28: Output of inclusive recurrence with call stack profiling 2

```
Timer1: 0 ms

Timer2: 6 ms

Call stack:
 Timer1;0.000000; ms
 Timer2;6.000000; ms
  Timer1;0.000000; ms
  Timer2;6.000000; ms
....
                     Timer1;0.000000; ms
                     Timer2;1.000000; ms
                      Timer1;0.000000; ms
                      Timer2;0.000000; ms
                       Timer1;0.000000; ms
                       Timer2;0.000000; ms
                        Timer1;0.000000; ms
                        Timer2;0.000000; ms
```

Without the call stack tracking:

Listing 2.29: Output of inclusive recurrence without call stack profiling 2

```

Timer1: 0 ms

Timer2: 0.2 ms
```

Relative speedup is:

- For timer_1 is not defined.
- For timer_2 is 96.6

## 2.5 Timers precision and performance impact

During this measurement a problem arises - actual granularity of <chorno> under Windows is close to 1 ms [6] [7]. This is a well-known compiler problem, and there are several ways to fix this:

- Use different timers, like boost::chrono, MPI or OMP get_wtime() (Windows and Linux), gettimeofday() - systime (Unix/POSIX)
- Use performance counters like QPC [8] (Windows only) or RDTSC [9] (Windows and linux. All x86 CPU )
- Use external libraries like likwid [10] or TAU [11]

After reimplementing macro defines to use QPC, measured data for empty recurrence changed greatly.

Without call stack tracking:

Listing 2.30: Output of inclusive recurrence without call stack profiling 3

```
Timer1: 1.74898e-005

Timer2: 0.357417
```

With call stack tracking:

Listing 2.31: Output of inclusive recurrence with call stack profiling 3

```
Timer1: 2.86195e-005

Timer2: 0.975421

Call stack:
 Timer1;0.000446; ms
 Timer2;0.975421; ms
  Timer1;0.000000; ms
  Timer2;0.974975; ms
....
                  Timer1;0.000892; ms
                  Timer2;0.000892; ms
                   Timer1;0.000000; ms
                   Timer2;0.000000; ms
```

Here the granularity is much higher and allows to profile less time-consuming regions of code more precisely. But question about cost of call stack tracking remains.

In order to determine cost of calling the measurement system one more test was written:

Listing 2.32: Test function for determining cost of timers calls

```
void TestTimer(int n )
{
        ProfilingTimers::_timer1.Start();
        for (int i = 0; i < n; i++)
        {
                ProfilingTimers::_timer2.Start();
                ProfilingTimers::_timer2.Stop();
        }
        ProfilingTimers::_timer1.Stop();
}
```

By supplying large enough n one can find the delta time for different timer types. The cost of Start() and Stop() calls can be determined by dividing outer timer (_timer1) value by number of iterations:

$$cost = \frac{timer\_val}{n}$$

| Timer name | n | | | |
|---|---|---|---|---|
| | $10^3$ | $10^4$ | $10^5$ | $10^6$ |
| Chrono | $0$ | $10^{-5}$ | $2 * 10^{-6}$ | $2.3 * 10^{-6}$ |
| QPC | $4.85 * 10^{-6}$ | $2.45 * 10^{-6}$ | $2.4585 * 10^{-6}$ | $2.46285 * 10^{-6}$ |
| WIN_TIME | $0$ | $1 * 10^{-5}$ | $3 * 10^{-6}$ | $2.9 * 10^{-6}$ |
| MPI_TIME | $2.05 * 10^{-5}$ | $2.08 * 10^{-5}$ | $2.08 * 10^{-6}$ | $2.07 * 10^{-6}$ |
| RDTSC | $2.46965 * 10^{-6}$ | $2.45566 * 10^{-6}$ | $2.46347 * 10^{-6}$ | $2.46807 * 10^{-6}$ |

Table 2.1: Cost, in milliseconds

This table shows that calling Start()/Stop() routine takes around 2.5 nanoseconds. This leads to conclusion that overhead caused by using this Timer class is neglectable.

Second run, with call stack collection enabled, adds roughly 4 nanoseconds, so total cost of Start()/Stop() routine is about 7 ns.

## Timer granularity

Timer granularity denotes the minimum possible time interval that can be measured, or timer resolution. Information about resolution is essential in order to reliably read and understand measured values.

For this purpose special test was written:

Listing 2.33: Test function for determining timer granularity

```
double FindMinInterval()
{
        TIMER_TIME_TIMEPOINT_TYPE _time_start = TIMER_NOW;
        TIMER_TIME_TIMEPOINT_TYPE _time_stop = TIMER_NOW;
        while (_time_stop == _time_start)
        {
                _time_stop = TIMER_NOW;
        }
        TIMER_TIME_STORE_TYPE diff = TIMER_GET_TIME_STORE(_time_stop -_time_start);
        return TIMER_GET_TIME_MILI(diff);
}
```

This allows to calculate the minimum time period that can be measured by specific timer and reveals granularity of specific timing method.

Resulting data is given in the next table:

| Timer | Interval |
|---|---|
| Chrono | $1$ |
| QPC | $4.462 * 10^{-4}$ |
| WIN_TIME | $1$ |
| MPI_TIME | $4.462 * 10^{-4}$ |
| RDTSC | $2.52174 * 10^{-5}$ |

Table 2.2: Timer granularity, ms

From this table it is clear that windows time_now(), std::chrono ( VS2012 ) has only 1 ms resolution. MPI and QPC have the same resolution [3] and RDTSC has the best resolution - 25 nanoseconds.

RDTSC is the best option for time measurement on x86 architecture, it is available on every modern x86-64 CPU. But values produced by this mechanism are frequency - dependent, so there are problems with using this on CPU with dynamic frequency. However, if frequency is constant during measurement this still will produce correct results. Another option is to take CPU frequency at the moment of measurement and convert it to time (ms, $\mu s$ ...).

For PowerPC there is different option - Time Base Register [12] .

## 2.6 Output format for profile

Since measured data will be automatically processed in the future , there should be some format for this. In this thesis Coma Separated Values (CSV [13]) format with semicolon as separator was selected for this purpose. This allows to have data which is really easy to parse and postprocess using different tools.

Example for timer:

Listing 2.34: Example of CSV for timers

```
NAME;X;Y
```

Here NAME is the timer name, X is the total time measured in ms and Y is the mean time for this timer.

Example for call stack:

Listing 2.35: Example of CSV for call stack entry

```
INDENT;NAME;X
```

Here INDENT is the call stack level, NAME is the timer name, X is the total time measured by this call entry.

## 2.7 Gathering measured data across the nodes

Since measured applications are used in supercomputing, there should be some mechanism to collect the data across the nodes. Here MPI is used for this purpose, and the functions like "Print-

[3] In MPI timer description it is said that it uses the best timer available. The reason why QPC and MPI resolutions are the same is that Microsoft MPI was used.

AllTimersToFileGlobal( std::string name )" are using MPI_Send/MPI_Recieve to collect data at the master thread. After collecting it can be saved for further processing.

From here and in future measurements, we will use application generated by Exastencil meta-compiler.

Here there is an example profile for application with two threads.

Listing 2.36: Example output for timers in CSV

```
setupWatch;138.008;138.008;stopWatch;14431.8;370.047;
        timeToSolveWatch;14858.9;14858.9;
setupWatch;121.007;121.007;stopWatch;14348.8;367.919;
        timeToSolveWatch;14859.9;14859.9;
```

And call stack for this profile:

Listing 2.37: Example output for callstack entry in CSV

```
0;setupWatch;138.008;0;timeToSolveWatch;14858.9;
        1;stopWatch;14431.8;
0;setupWatch;121.007;0;timeToSolveWatch;14859.9;
        1;stopWatch;14348.8;
```

Call stack is represented in a tree form: every indent shows the depth of call entry, and order imposes child-parent relation. Call stack profile from 2.37 is decoded as following:

- setupWatch is the first entry from zero level, measured time 138.008 ms.
- timeToSolveWatch is the second entry on zero level,measured time 14858.9 ms.
- stopWatch is first entry from level one and is a child call from previous zero-level entry, e.g. timeToSolveWatch. It has inclusive time of 14431.8 ms.

We will talk more about data postprocessing in section "Data visualisation" of this chapter.

## 2.8 Data visualization

In selected scripting environment the matplotlib [14] library is used to visualize measured profiles. It allows to visualize numerical data in 2D and 3D. This package is often used for scientific data visualization [5] and data visualization for publications, presentations, theses.

In order to investigate measured data visually the Python program was written and it allows us to:

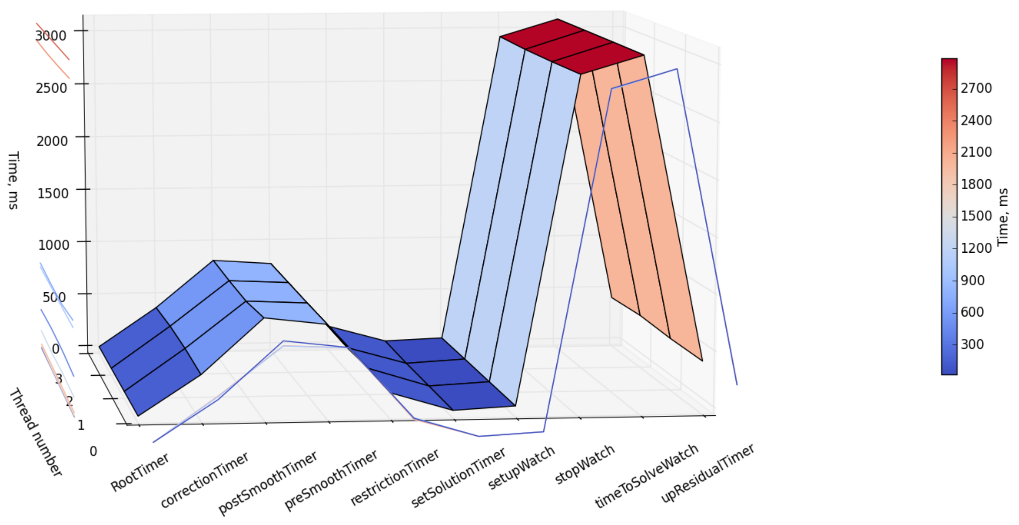
- Show 3D plot of all timers from all threads
- Show per thread and per timer projection plot
- Plot call stack

Core idea of profile visualization is that data, presented in performance profile and call stack profile, has either flat sequential (data from performance profile) or tree-like (call stack) structure. Depending on data format script dynamically uses appropriate way to visualize them.

Visual representation helps to see behaviour of an application in a more convenient way than raw data.

On figure [2.1] an example of 3D application profile is shown.

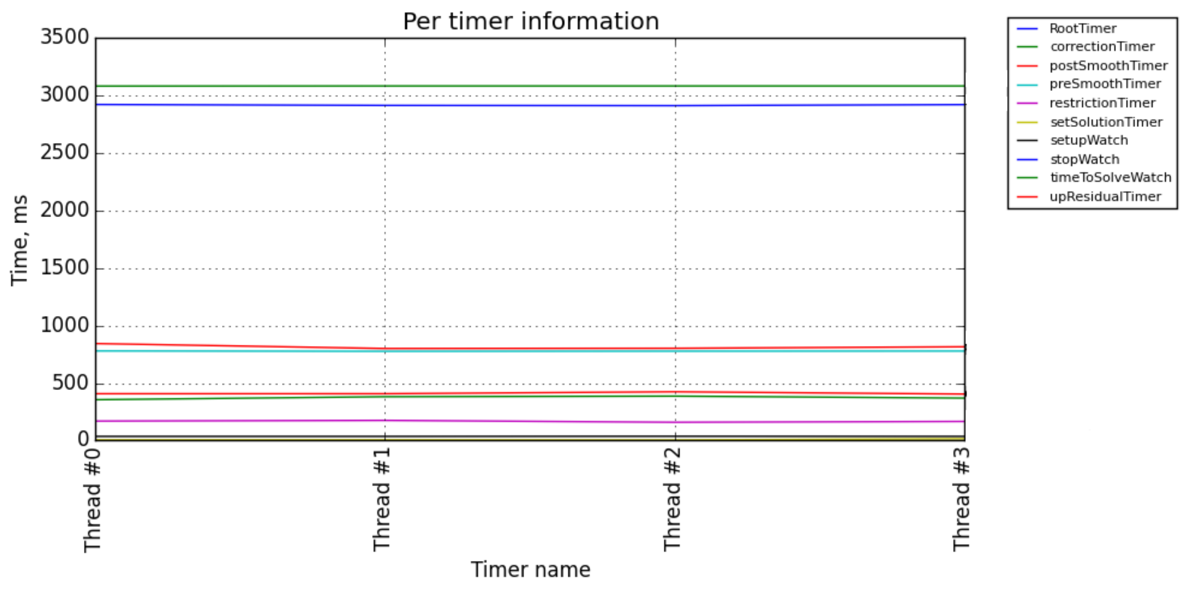
Figure 2.1: 3D view of application profile



Here on the "X" axis there are timers names, used in profiled application. They are output in the same order as they were first called. Thread number is on the "Y" axis [4]. "Z" axis values are total time measured by the corresponding timer. This is plotted as surface for better perception.

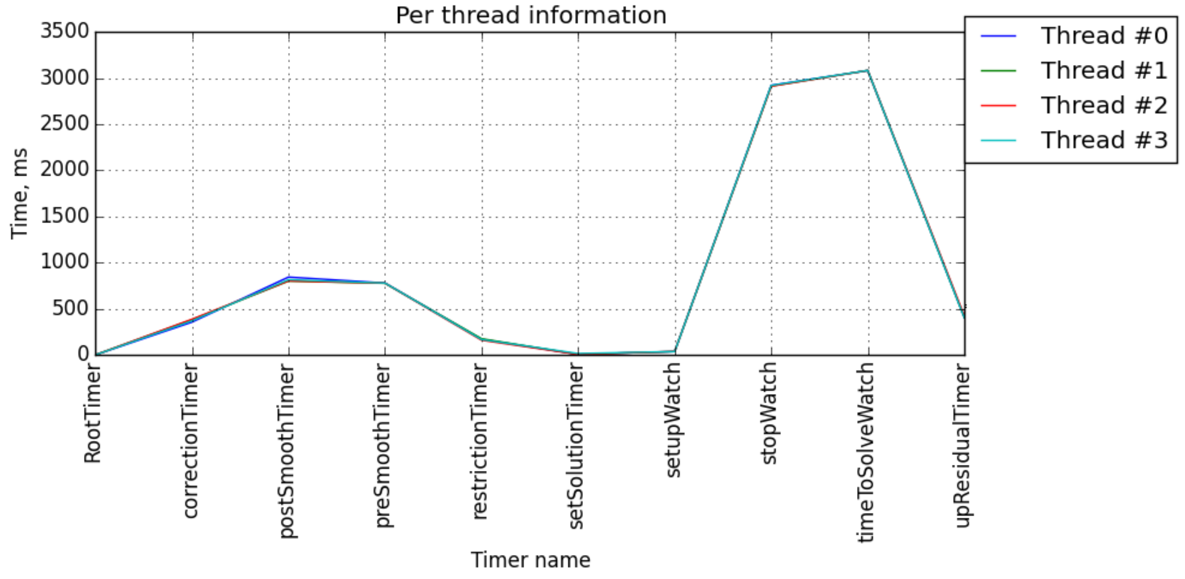
The next two plots are showing projection of this data on z-y [2.2] plane and on z-x [2.3] plane.

Figure 2.2: 2D z-y projection



[4]profiled application is 4-threaded

Figure 2.3: 2D z-x projection



Matplotlib allows to zoom in/out the plot. On click user can receive the line description [2.4], [2.5].

Figure 2.4: 2D z-y projection on click example

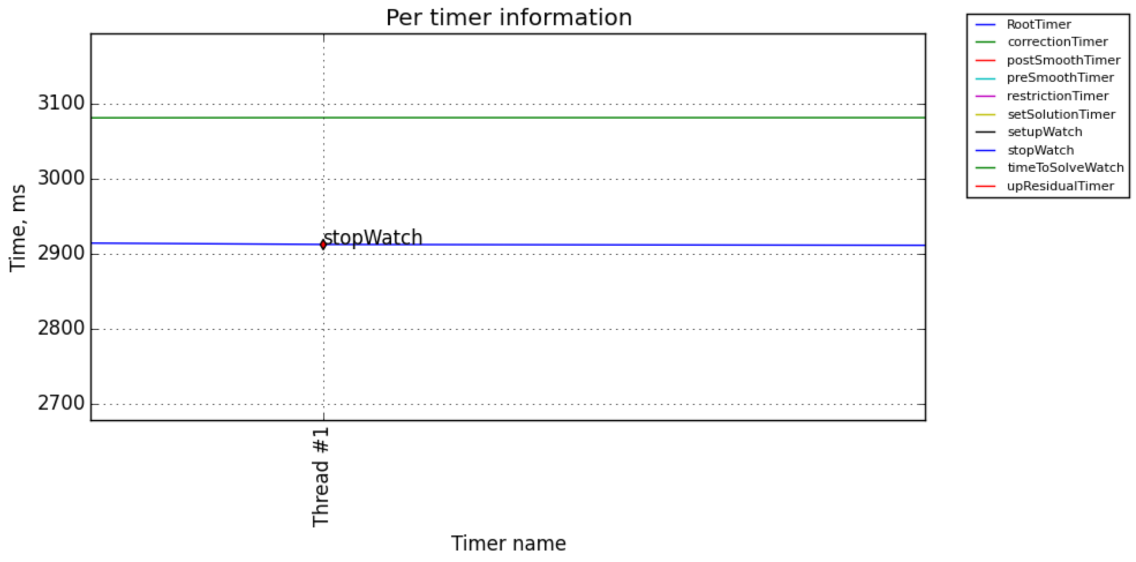
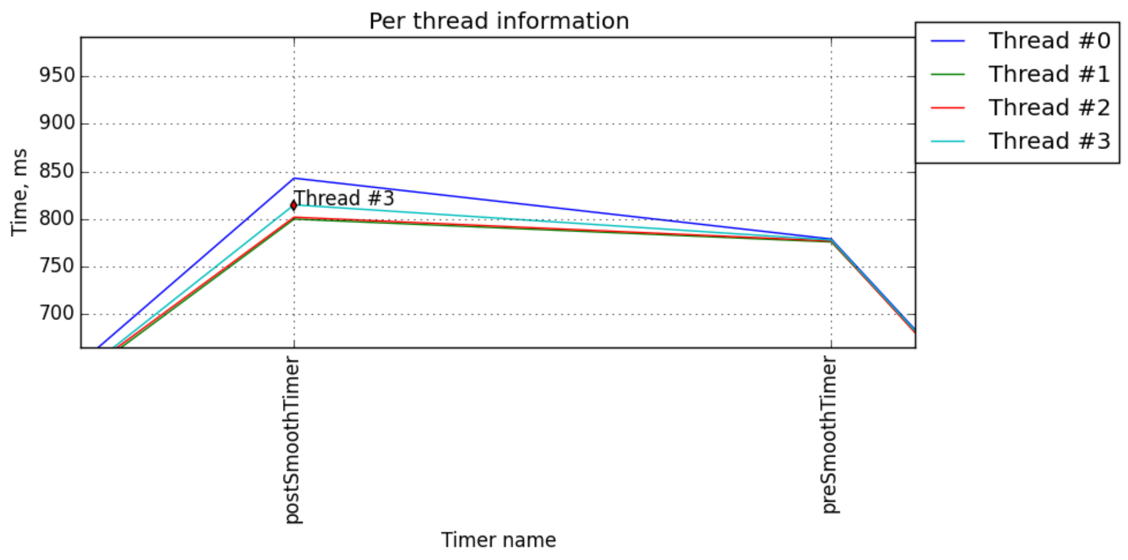


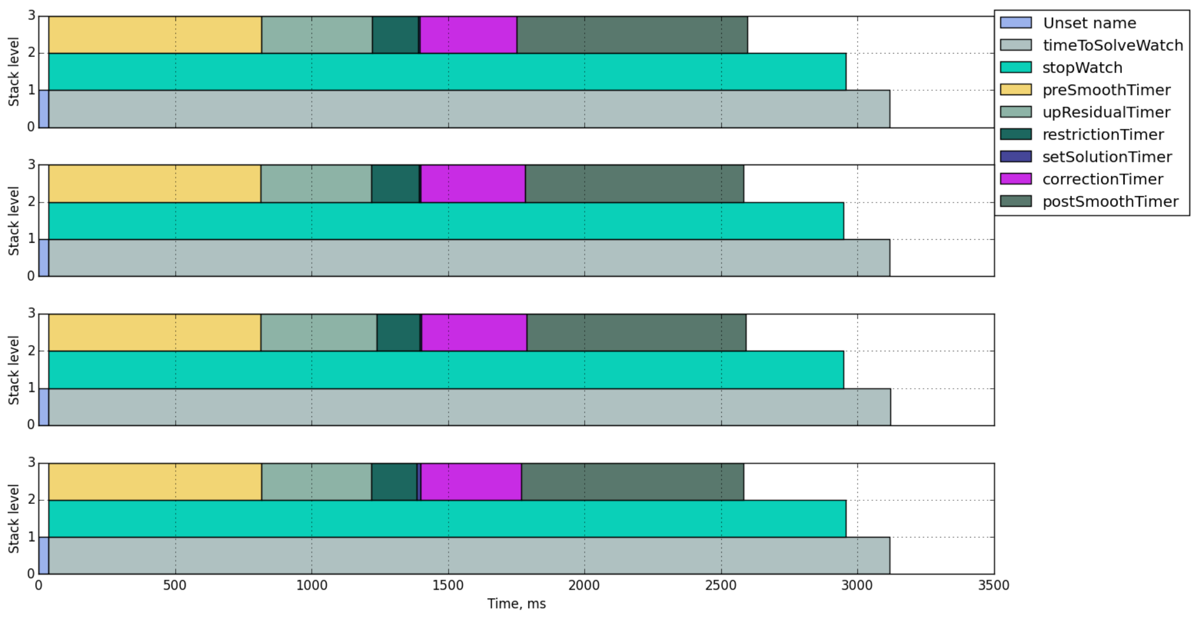
Figure 2.5: 2D z-x projection on click example



With call stack profile it was harder to find a way to visualize it in a way other that tree-like. We usually inspect callstack by outputting the values in text or tree form. However, I wanted to visualize it in a fully graphical way and the way to do it was found after observing IncrediBuild run-time output [15].

On plot [2.6] we can observe call stack output.

Figure 2.6: 2D plot of call stack for every MPI thread



Every line represents a level of inclusion and call sequence on this level (within its parent) and its length (time). Also, it allows to see the difference in execution between different threads. This information may be helpful to find bottlenecks in communication or computation power utilization by different nodes.

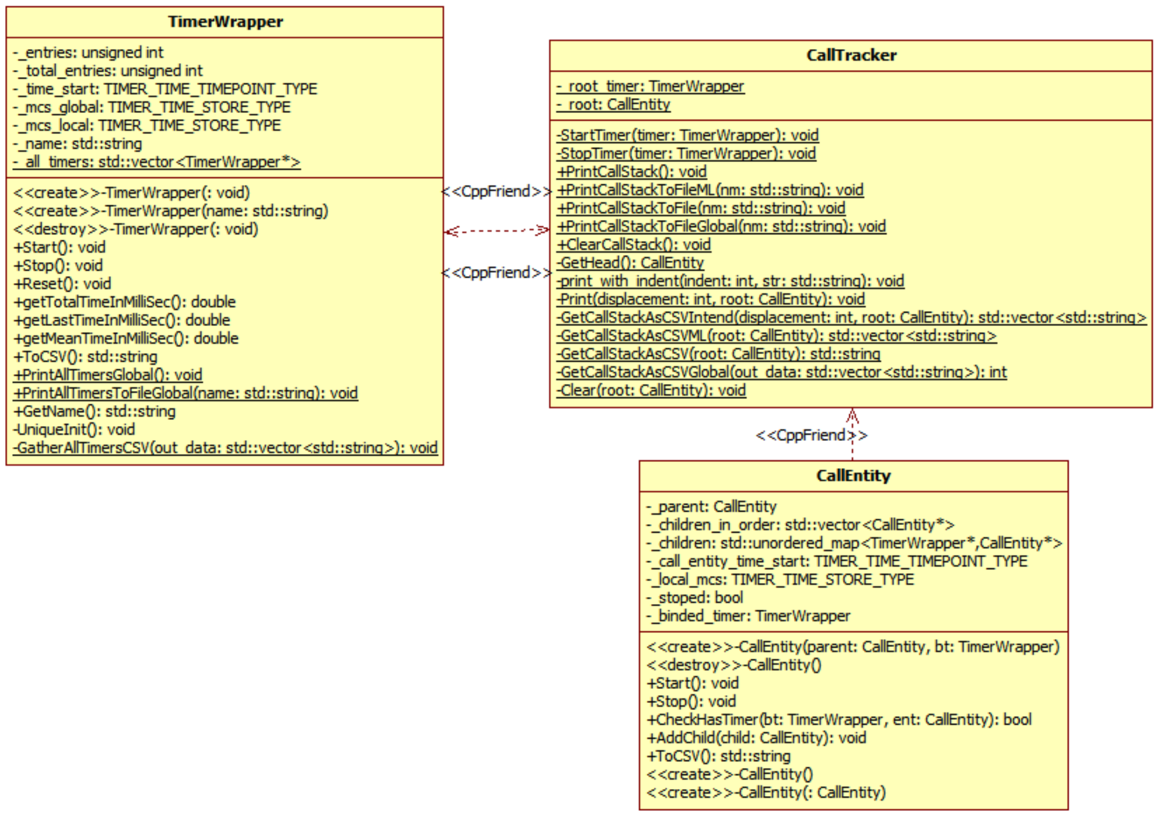
## 2.9 Summary

At this point the measurement infrastructure for performance evaluation has been created and tested. The impact of measurement system for heavy enough code regions is neglectable. The macro defines allow to easily modify code in case of automatic generation. This minimizes time to change from one measurement system to another in case of manual code change. CSV format will be used in the next chapters of this thesis. Call stack tracking allows to produce more detailed profile with relatively small overhead[5].

Visualization script allows to visually investigate measured profile for better understanding of program behaviour.

The final class diagram for Stopwatch, Call entity and Call Tracker is presented on figure [2.7]

Figure 2.7: UML diagram for Stopwatch


TimerWrapper
-_entries: unsigned int
-_total_entries: unsigned int
-_time_start: TIMER_TIME_TIMEPOINT_TYPE
-_mcs_global: TIMER_TIME_STORE_TYPE
-_mcs_local: TIMER_TIME_STORE_TYPE
-_name: std::string
-_all_timers: std::vector<TimerWrapper*>
<<create>>-TimerWrapper(: void)
<<create>>-TimerWrapper(name: std::string)
<<destroy>>-TimerWrapper(: void)
+Start(): void
+Stop(): void
+Reset(): void
+getTotalTimeInMilliSec(): double
+getLastTimeInMilliSec(): double
+getMeanTimeInMilliSec(): double
+ToCSV(): std::string
+PrintAllTimersGlobal(): void
+PrintAllTimersToFileGlobal(name: std::string): void
+GetName(): std::string
-UniqueInit(): void
-GatherAllTimersCSV(out_data: std::vector<std::string>): void
<<CppFriend>>
<<CppFriend>>
CallTracker
-_root_timer: TimerWrapper
-_root: CallEntity
-StartTimer(timer: TimerWrapper): void
-StopTimer(timer: TimerWrapper): void
+PrintCallStack(): void
+PrintCallStackToFileML(nm: std::string): void
+PrintCallStackToFile(nm: std::string): void
+PrintCallStackToFileGlobal(nm: std::string): void
+ClearCallStack(): void
-GetHead(): CallEntity
-print_with_indent(indent: int, str: std::string): void
-Print(displacement: int, root: CallEntity): void
-GetCallStackAsCSVIntend(displacement: int, root: CallEntity): std::vector<std::string>
-GetCallStackAsCSVML(root: CallEntity): std::vector<std::string>
-GetCallStackAsCSV(root: CallEntity): std::string
-GetCallStackAsCSVGlobal(out_data: std::vector<std::string>): int
-Clear(root: CallEntity): void
<<CppFriend>>
CallEntity
-_parent: CallEntity
-_children_in_order: std::vector<CallEntity*>
-_children: std::unordered_map<TimerWrapper*,CallEntity*>
-_call_entity_time_start: TIMER_TIME_TIMEPOINT_TYPE
-_local_mcs: TIMER_TIME_STORE_TYPE
-_stoped: bool
-_binded_timer: TimerWrapper
<<create>>-CallEntity(parent: CallEntity, bt: TimerWrapper)
<<destroy>>-CallEntity()
+Start(): void
+Stop(): void
+CheckHasTimer(bt: TimerWrapper, ent: CallEntity): bool
+AddChild(child: CallEntity): void
+ToCSV(): std::string
<<create>>-CallEntity()
<<create>>-CallEntity(: CallEntity)

[5]A new paper will be published soon, in which call stack accuracy with compiler supported sampling is investigated. [16].

# Chapter 3

# Automatic code generation, compilation and execution

*"Artificial living plants"*

— Edward F. Moore

The problems further discussed in this thesis are batch code generation, measurement and processing. In order to solve this a special Python script was written, which allows to specify parameters that should be used for code generation, compilation and execution.

## 3.1 Overview

As mentioned before, code to be measured is produced by Exastencils code generator (metacompiler).

Exastencil code generator is the metacompiler which compiles from domain specific language (DSL) to general purpose language (C++).

In this thesis Exastencil metacompiler (code generator) is treated as a black box, generating the target source code based on the input files. One input file is "Layer4.exa" containing the description of the problem in domain specific language. Two other files provide variables to the code generator. These variables provide information about program specialization in order to produce one specific C++ program.

Problem described in DSL is constant and does not change in this thesis. It is a simple multigrid metaprogram, that stops execution once it converges to solution - residual becomes smaller than some threshold (multigrid methods are out of scope of this thesis, so there is no introduction to it).

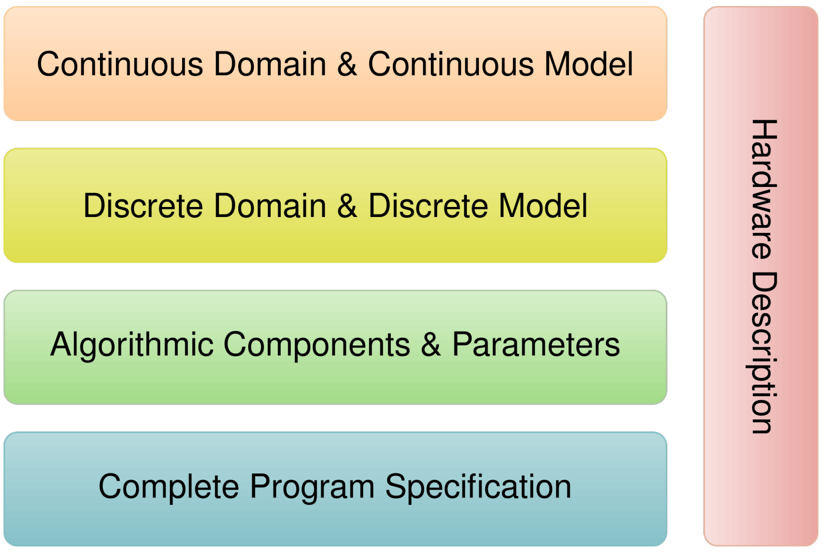
DSL used in Exastencils is called ExaSlang [17]

## 3.2 Exastencil metacompiler

Exastencill metacompiler is a part of Exastencils project, and allows to compile level 4 DSL code to C++. Besides "Layer4.exa" input file, two other are Knowledge(*.knowledge) and Settings (*.settings) files. Metacompiler is written in Scala [18] language. Additionally, we provide a Scala generation function that sets up a Stopwatch construct in target language (C++).

The code generating flow is presented on figure [3.1].

Figure 3.1: Exastencils chart 1


Continuous Domain & Continuous Model
Discrete Domain & Discrete Model
Algorithmic Components & Parameters
Complete Program Specification
Hardware Description

Complete program specification (layer4) is the input to metacompiler.

Additional variables supplied to metacompiler are defined in the layer 3 and included in layer 4 of the generation flow. A more detailed description is given in this figure [3.2].

Figure 3.2: Exastencils chart 2 [1]

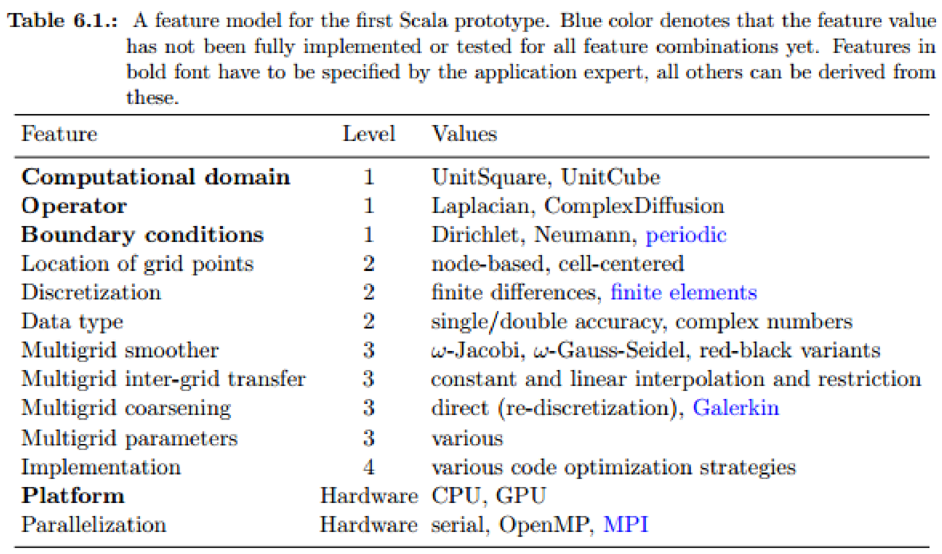


**Table 6.1.:** A feature model for the first Scala prototype. Blue color denotes that the feature value has not been fully implemented or tested for all feature combinations yet. Features in bold font have to be specified by the application expert, all others can be derived from these.

| Feature | Level | Values |
|---|---|---|
| **Computational domain** | 1 | UnitSquare, UnitCube |
| **Operator** | 1 | Laplacian, ComplexDiffusion |
| **Boundary conditions** | 1 | Dirichlet, Neumann, periodic |
| Location of grid points | 2 | node-based, cell-centered |
| Discretization | 2 | finite differences, finite elements |
| Data type | 2 | single/double accuracy, complex numbers |
| Multigrid smoother | 3 | $\omega$-Jacobi, $\omega$-Gauss-Seidel, red-black variants |
| Multigrid inter-grid transfer | 3 | constant and linear interpolation and restriction |
| Multigrid coarsening | 3 | direct (re-discretization), Galerkin |
| Multigrid parameters | 3 | various |
| Implementation | 4 | various code optimization strategies |
| **Platform** | Hardware | CPU, GPU |
| Parallelization | Hardware | serial, OpenMP, MPI |

Parameters mentioned on layer 3 are the ones which are supplied to metacompiler with layer 4 program description.

Layer 4 variables are stored in Knowledge, example is shown here:

Listing 3.1: Example Knowledge file

```
targetCompiler                              = "GCC"
simd_instructionSet                         = "SSE3"
```

```
dimensionality                          = 3

maxLevel                                        = 6
comm_strategyFragment           = 6
domain_numBlocks_x                      = 3
domain_numBlocks_y                      = 3
domain_numBlocks_z                      = 1
domain_numFragsPerBlock_x       = 3
domain_numFragsPerBlock_y       = 3
domain_numFragsPerBlock_z       = 1
domain_fragLength_x                     = 1
domain_fragLength_y                     = 1
domain_fragLength_z                     = 1

l3tmp_genAdvancedTimers         = true
advTimer_timerType = "Chrono"
advTimer_enableCallStacks = false
l3tmp_smoother = "GS"
l3tmp_numPre = 3
l3tmp_numPost = 3
```

Explanation:

- "targetCompiler" is the compiler name that will be used to compile generated program (MSVC, GCC or IBMXL).
- "simd_instructionSet" is the instruction set used for computation speed up (based on data parallelism ). Possible values -SSE3, AVX, AVX2.
- "dimensionality" sets space dimensionality for the problem - 1D, 2D, 3D.
- "maxLevel" sets the finest level possible for multigrid. Every additional level increases problem size by 2.
- "comm_strategyFragment" specifies whether communication is only performed along coordinate axis or to all neighbors. Possible values - 6 or 26.
- "domain_numBlocks_*" - specifies number of blocks per dimension - one block will usually be mapped to one MPI thread.
- "domain_numFragsPerBlock_*" specifies the number of fragments in each block per dimension - is usually one or represents the number of OMP threads per dimension.
- "domain_fragLength_*" sets the length of each fragment per dimension - this will either be one or specify the length in unit-fragments, i.e. the number of aggregated fragments per dimension.
- "l3tmp_genAdvancedTimers" notifies metacompiler that we want to use timers, described in Chapter 2.
- "advTimer_timerType" sets the time measurement system to be used with timers from Chapter 2.
- "advTimer_enableCallStacks" allows to enable/disable call stack tracking.
- "l3tmp_smoother" specifies the smoother type, used in multigrid (smoothing high-frequency errors). Can be "GS" (Gauss-Seidel iteration), "Jac" (Jacobi iteration), or "RBGS" (Red-Black Gauss-Seidel).
- "l3tmp_numPre" and "l3tmp_numPost" specifies how many iterations of smoothing will be performed within one multigrid cycle (at the beginning and at the end, respectively).

Knowledge file is to be generated automatically. Knowledge file generation together with automatic compilation and execution are discussed in the following sections.

## 3.3 Batch code generation

### Input data for batch code generation

In order to generate Knowledge file automatically, there should be some way of categorizing data, that is defined in there. Taking into account that in future these variables can (and will) be treated as genes for genetic algorithm, special categorizing was developed.

Knowledge file variables can be divided into 3 main categories:

- Range variables, such as # of blocks or # of smoothing steps
- List variables, such as l3tmp_smoother
- Constants, such as targetCompiler or simd_instructionSet

In order to specify them, special project file was designed, allowing to set these and many other variables.

#### Range variables

Range variables (or range genes) are genes which can get any integer value in a given range. The following listing shows an example of defining such a variable:

Listing 3.2: Example range gene

```
["domain_numBlocks_x",2,2]
```

The first entry is a variable name, the second one is the starting value and third one shows how many values are in the range. Hence, the example above produces two possible variables "2" and "3".

The set of such definitions is stored in a special array - "range_genes_def":

Listing 3.3: Set of range genes

```
range_genes_def =
["domain_numBlocks_x",1,2],
["domain_numBlocks_y",1,2],
["domain_numBlocks_z",1,2],
["domain_numFragsPerBlock_x",1,2],
["domain_numFragsPerBlock_y",1,2],
["domain_numFragsPerBlock_z",1,2],
["domain_fragLength_x",1,1],
["domain_fragLength_y",1,1],
["domain_fragLength_z",1,1],
["l3tmp_numPre",1,4],
["l3tmp_numPost",1,4],
["maxLevel", 8, 1]#finest level
]
```

#### List variables

List variables (or list genes) are genes which can only get values specified in gene description. The following listing shows an example of defining such a variable:

Listing 3.4: Example list gene

```
["l3tmp_smoother", ["\"GS\"","\"Jac\"","\"RBGS\""] ]
```

The first entry is name, the second is the array of possible values for this gene.

The set of such definitions is stored in a special array - "list_genes_def":

Listing 3.5: Set of list genes

```
list_genes_def = [
["comm_strategyFragment",[6,26]],
["l3tmp_smoother", ["\"GS\"","\"Jac\"","\"RBGS\""] ]
]
```

### Constants

Constants (or constant genes) are variables which cannot change their value during any phase of code generation. Generally, they specify platform- or environment-dependent parameters, for example, the compiler type:

Listing 3.6: Example of constant gene 1

```
["targetCompiler","\"GCC\""]
```

Or SIMD instruction set to use:

Listing 3.7: Example of constant gene 2

```
["simd_instructionSet","\"SSE3\""]
```

The set of such definitions is stored in a special array - "const_genes":

Listing 3.8: Set of constant genes

```
const_genes = [
["targetCompiler","\"GCC\""],
["simd_instructionSet","\"SSE3\""],
["dimensionality",3],
["l3tmp_genFunctionBC", "false"],#only for 2D
["l3tmp_genAdvancedTimers","true"],
["advTimer_timerType","\"Chrono\""]
]
```

### Program specification

One particular program (or unit, or genome) is specified as a combination of these three sets. By picking particular values for range and list genes and combining them with constant genes one receives a complete set of parameters to be sent to metacompiler.

This is the key idea behind batch code generation.

## Automatic batch code generation

Now that all possible variables for each entry of each genome description category are defined, we can apply full Cartesian product to it, treating every list and range genes as separate dimensions.

In order to automate this process we have written a Python script. It will be generating all possible combinations of parameters mentioned above. The input parameter for it is a special project file, where all (range, list, and constant) variables are defined alongside with additional parameters like "out_path" (for storing additional data generated during script execution) or "lib_path" (for path to special storage of generated programs).

With a small helper function we convert genes into Knowledge file:

Listing 3.9: Writing genes to *.knowledge file

```
def WriteKnowledge(path, genome):
        file = open(path, 'w+')
        for gene in vars.const_genes:
                file.write("%s␣=␣%s\n" % (gene[0],gene[1]))
        file.write("%s\n" % "")

        for i in range(len(genome)):
                file.write("%s␣=␣%s\n" % (non_const_gene_names[i],genome[i]))
....
```

Array "non_const_gene_names" stores names of range and list genes. The order of genes always corresponds to the name sequence within the script. "vars" object is an imported project file, that allows to access variables in the following way:

Listing 3.10: Accessing project file data

```
vars.const_genes
```

Settings file stores output path where code should be generated and some additional settings like "failOnConstraint" (if "true" imposes additional restriction from compiler side[1]).

## Code generation

After writing Knowledge and Settings files to target folder, Exastencils metacompiler can be called. In order to do so we need to navigate to compiler folder and call it with generated files as arguments:

Listing 3.11: Metacompiler call

```
def GenerateCodeForUnit(unit, path_to_compiler):
        if unit == None:
                return None

        tpath = vars.lib_path + FunctionLibrary.UnitToPath(unit);

        if not os.path.exists(tpath):
                os.makedirs(tpath)
        if os.path.isfile(tpath + "/" + failed_marker):# failed generation
                pass#do nothing
        elif os.path.isfile(tpath+"/Makefile"):# already generated
                return [tpath, FunctionLibrary.GetNPforUnit(unit,non_const_gene_names)]
        else:
                cwd = os.getcwd()
                os.chdir(path_to_compiler)
                try:
                        command = "java␣−cp␣compiler.jar␣Main␣" + tpath+"/example.settings"+ "␣"
+ tpath +"/example.knowledge"
                        with open(os.devnull, "w") as f:
                                sp.call( command.split(), stdout=f)
                        if os.path.isfile(tpath+"/Makefile"):
                                return[tpath, FunctionLibrary.GetNPforUnit(unit,
                                                                              non_const_gene_names)]
                        else:#means NO MAKEFILE, failed project
                                WriteNUMB(tpath)
                finally:
                        os.chdir(cwd)

        return None
```

[1]Generally the metacompiler will try to fix incorrect Knowledge file if it is possible. But since every variable in this file will be used as genes for Genetic Algorithm, we do not want to have any kind of auto-correction.

Within this function all consistency checks are performed. We either get a path to executable and a number of MPI threads to be used to call an executable or "None" object, if generation failed at any step.

### Generated programs library

In order to preserve already generated and compiled programs special folder structure is presented as special storage.

It is automatically defined by range and list definitions provided in project file, for instance, one possible unit from sets defined in 3.3, 3.5 can be represented as follows:

```
/1/1/1/1/1/1/1/1/1/3/2/8/6/GS

/domain_numBlocks_x/domain_numBlocks_y/domain_numBlocks_z/domain_numFragsPerBlock_x/
domain_numFragsPerBlock_y/domain_numFragsPerBlock_z/domain_fragLength_x/
domain_fragLength_y/domain_fragLength_z/l3tmp_numPre/l3tmp_numPost/maxLevel/
comm_strategyFragment/l3tmp_smoother
```

First line consists of values of corresponding variables (genes) specified in the second line. The path for this unit looks like this: "*lib_path*/_1/_1/_1/_1/_1/_1/_1/_1/_1/_3/_2/_8/_6/_GS/"

Together with consistency checking ( special check at the beginning of a script, where it checks whether folder structure represents Knowledge file stored in this path). This allows fast navigation and reuse of already generated code.

### Example project file

The simplest project file that gives required information for generate-compile-run (GCR) routine is presented in listing [3.12] (later on project file will get more variables as more features will be added).

Listing 3.12: Example project file

```
range_genes_def =       [
["domain_numBlocks_x",2,1],
["domain_numBlocks_y",2,1],
["domain_numBlocks_z",1,1],
["domain_numFragsPerBlock_x",1,1],
["domain_numFragsPerBlock_y",1,1],
["domain_numFragsPerBlock_z",1,1],
["maxLevel", 7, 1]
]

list_genes_def = [
["comm_strategyFragment",[6]],
["l3tmp_smoother", ["\"Jac\"","\"GS\""] ]
]

const_genes = [
["targetCompiler","\"GCC\""],
["simd_instructionSet","\"SSE3\""],
["dimensionality",3],
["l3tmp_genFunctionBC", "false"],
["l3tmp_genAdvancedTimers","true"]
]
out_path = '../path_to_output/'
lib_path = '../library/'
```

## Batch code compilation and execution

After the code is generated it can be compiled and executed producing profiles which can be used for visualisation or for further processing.

This part is simple, we are calling the "make" command and then executing compiled program by calling "mpirun -np X ./exastencils". This X depends on genes from genome like "domain_numBlocks_x", so this value is known in advance.

Listing 3.13: Execution flow

```
...
        #Compile
        CompileUnit(tpath, base_path_to_Script, lnch_data[1] )
        #Run
        if (vars.run_while_generating_or_genetics):
                file = StartMAKEEXECRoutin(tpath, str(lnch_data[1]) )
...
```

In code snippet [3.13] the compile-run flow is shown: CompileUnit(...) calls Make on generated code, then, if program should be run immediately after compilation, it is executed, and, thus, generates profile.

If end user wants to only generate and compile code and execute it later, it is also possible. Such an option is good to use on big clusters with limited time and pre-requested execution power. One can generate and compile it on the local machine (with access to file system on cluster) and request computation power only for the execution.

But the problem here is that this routine can fail at any step, so there should be some mechanism to make sure that if unit was failed at any point, it will not be used again. This allows to save total time spent within algorithm and notify other processing parts that there is no measured data and there is no way to obtain it. This is solved by checking for existing generated files at every step of pipeline. If expected file is missing the special marker file is written in directory of current processed unit.

### Restrictions and additional options

Taking into account that direct Cartesian product of genes can lead to situation where total number of threads exceeds available computation resources, there should be a way to impose restrictions.

This is done by setting additional variables to project file:

Listing 3.14: Per-thread restrictions

```
...
num_omp_per_mpi = [1,8]
num_mpi = [1,8]
num_total_threads = [1,8]
...
```

Here "num_omp_per_mpi" specifies the range for number of OpenMP threads per MPI thread, "num_mpi" sets range for number of MPI threads and "num_total_threads" specifies total maximum number of threads per program.

Additional parameters are:

- "cmp_path" - sets the absolute path to metacompiler.

- "run_while_generating_or_genetics" = True or False - if program should be executed after generation or during run of genetic algorithm.
- "generate_predifined" = True or False - if we want to generate code for predefined units, stored in "predifined_genes" array. Yet constant genes will be used for this, so only list and range genes should be specified here.
- "measured_timers_fname" = "TimersAll.msd" or any other ( defined by end user, should be same as used for saving output profile ) - defines file name of generated profile.
- "strong_scale" = 0 or $2^N$ ( where N denotes "maxLevel" ) - if 0, generation of all combination from range and list defines will be performed. If non-zero, problem size will be kept constant (if possible). This means that "maxLevel" from range variables will be ignored and computed value w.r.t. number of threads per dimension will be used instead.

These settings allows more flexible use of batch processing script.

## Script call

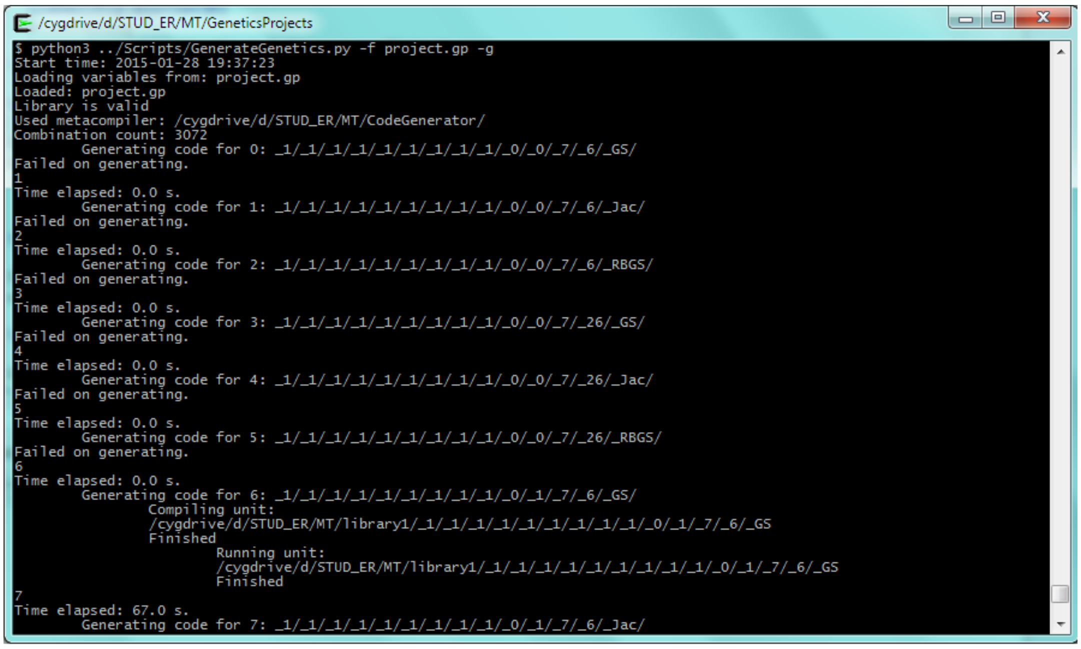
In order to execute script that performs automatic GCR routing with some project file, user should navigate to the folder that contains this file, and execute following command:

"Python3 PATH_TO_SCRIPTS/GenerateGenetics.py -f project.gp -g"

Here "PATH_TO_SCRIPTS" is relative or absolute path to folder, containing "GenerateGenetics.py" script, "-f project.gp" defines what project to load and "-g" key tells the script that user wants to generate all units w.r.t. definitions in project file. Python of version "3" or higher should be used, preferably 3.3+.

Example output is presented on figure [3.3].

Figure 3.3: Generation script output

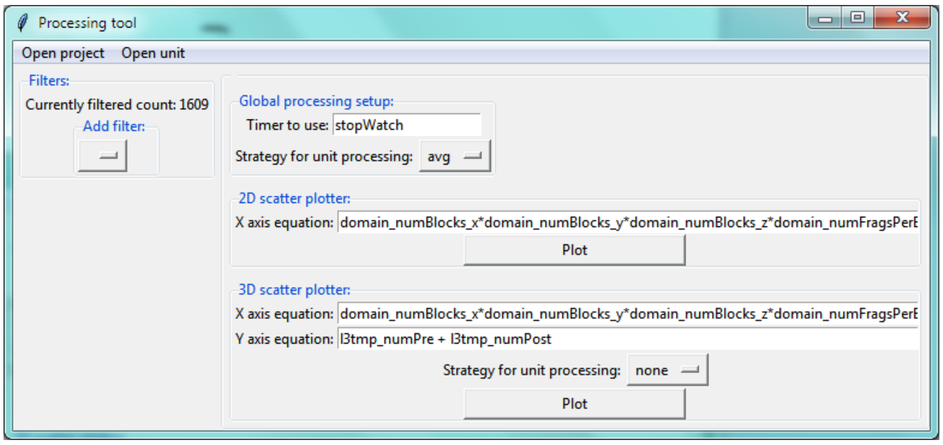


```
$ python3 ../Scripts/GenerateGenetics.py -f project.gp -g
Start time: 2015-01-28 19:37:23
Loading variables from: project.gp
Loaded: project.gp
Library is valid
Used metacompiler: /cygdrive/d/STUD_ER/MT/CodeGenerator/
Combination count: 3072
        Generating code for 0: _1/_1/_1/_1/_1/_1/_1/_1/_1/_0/_0/_7/_6/_GS/
Failed on generating.
1
Time elapsed: 0.0 s.
        Generating code for 1: _1/_1/_1/_1/_1/_1/_1/_1/_1/_0/_0/_7/_6/_Jac/
Failed on generating.
2
Time elapsed: 0.0 s.
        Generating code for 2: _1/_1/_1/_1/_1/_1/_1/_1/_1/_0/_0/_7/_6/_RBGS/
Failed on generating.
3
Time elapsed: 0.0 s.
        Generating code for 3: _1/_1/_1/_1/_1/_1/_1/_1/_1/_0/_0/_7/_26/_GS/
Failed on generating.
4
Time elapsed: 0.0 s.
        Generating code for 4: _1/_1/_1/_1/_1/_1/_1/_1/_1/_0/_0/_7/_26/_Jac/
Failed on generating.
5
Time elapsed: 0.0 s.
        Generating code for 5: _1/_1/_1/_1/_1/_1/_1/_1/_1/_0/_0/_7/_26/_RBGS/
Failed on generating.
6
Time elapsed: 0.0 s.
        Generating code for 6: _1/_1/_1/_1/_1/_1/_1/_1/_1/_0/_1/_7/_6/_GS/
                Compiling unit:
                /cygdrive/d/STUD_ER/MT/library1/_1/_1/_1/_1/_1/_1/_1/_1/_1/_0/_1/_7/_6/_GS
                Finished
                        Running unit:
                        /cygdrive/d/STUD_ER/MT/library1/_1/_1/_1/_1/_1/_1/_1/_1/_1/_0/_1/_7/_6/_GS
                        Finished
7
Time elapsed: 67.0 s.
        Generating code for 7: _1/_1/_1/_1/_1/_1/_1/_1/_1/_0/_1/_7/_6/_Jac/
```

## 3.4 Data visualisation

At this point one can define hundreds and thousands of units to be generated and executed, so there should be a way to inspect measured data. Possibility to do so is granted by another Python program.
It allows to load the project file, and, based on variables set, will generate interface and load measured data [3.4].

Figure 3.4: Processing tool



Currently filtered count: 1609

Timer to use: stopWatch

Strategy for unit processing: avg

X axis equation: domain_numBlocks_x*domain_numBlocks_y*domain_numBlocks_z*domain_numFragsPerE

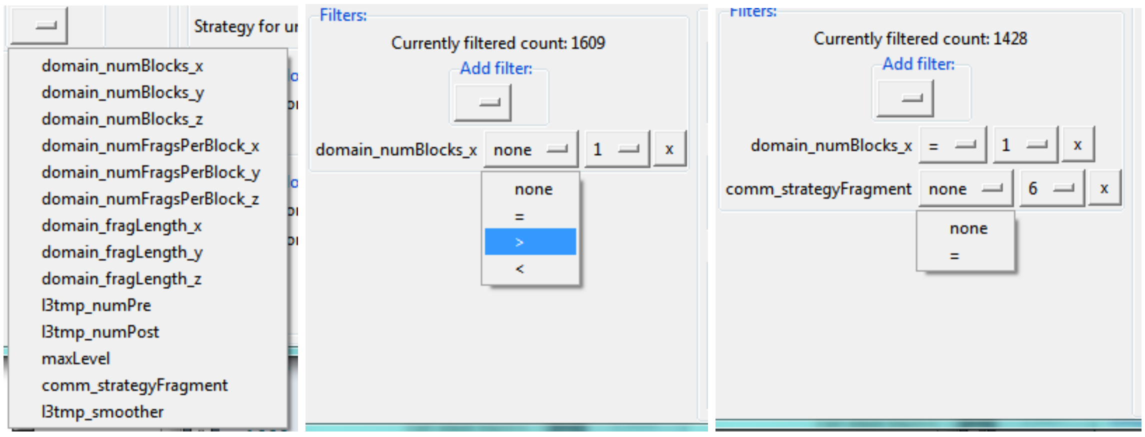
X axis equation: domain_numBlocks_x*domain_numBlocks_y*domain_numBlocks_z*domain_numFragsPerE

Y axis equation: l3tmp_numPre + l3tmp_numPost

Strategy for unit processing: none

In the filters frame user can add filters. These filters are based on range and list definitions from project file. Range filters can be set to filter values that are greater, equal or smaller than certain value (range of these values is taken form range in gene description). For list genes one can pick only equality. [3.5]

Figure 3.5: Processing tool filters



Currently filtered count: 1609

domain_numBlocks_x none 1 x

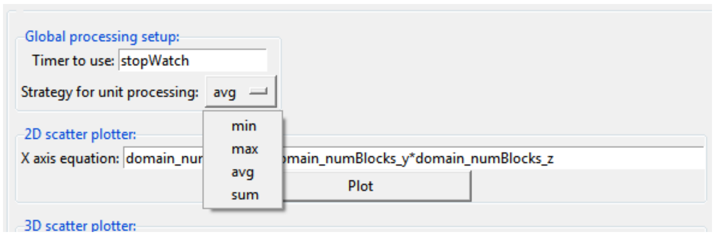
Currently filtered count: 1428

domain_numBlocks_x = 1 x

comm_strategyFragment none 6 x

On the "Global processing setup" frame should be defined timer name, which values should be used for post-processing, and the post-processing strategy [3.6].

Figure 3.6: Processing tool global settings



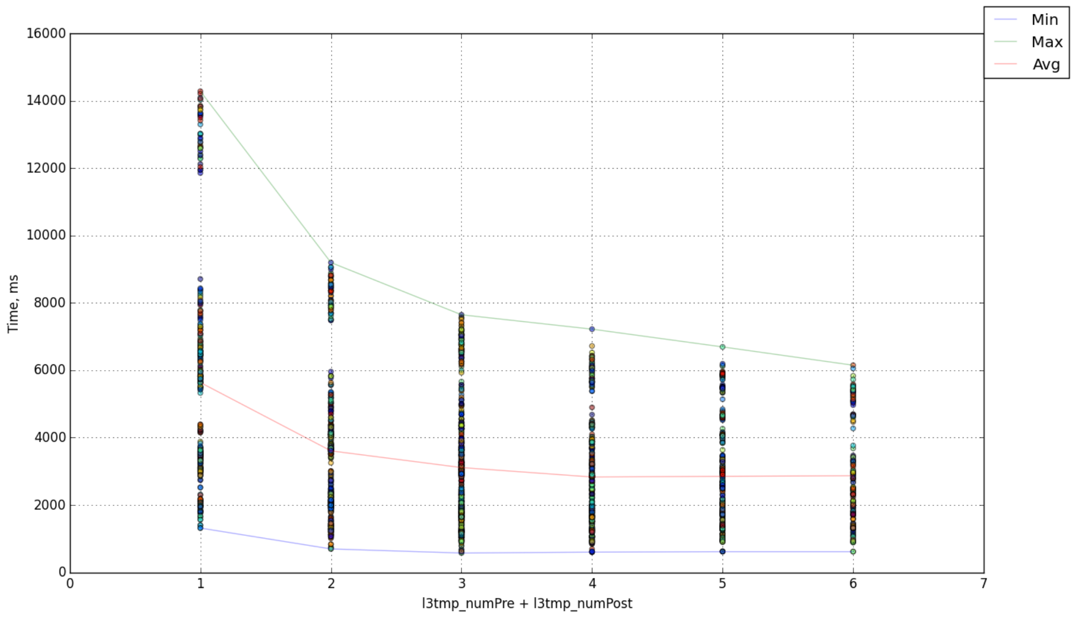
Options here are:

- "min" - pick minimal value across the threads
- "max" - pick maximal value across the threads
- "avg" - pick average value across the threads
- "sum" - use sum of all value across the threads

After this step every unit has a corresponding (double) value that can be visualized on a scatter plot. There is a lot of units, so there should be an option to assign some values for them on X-axis (in case of 2D) or on X-Y plane (3D).
In case of 2D for this purpose there are editable fields where user can specify equation for assigning positions on X-axis, e.g. the total number of threads or problem size.

For instance, with equation "l3tmp_numPre + l3tmp_numPost" one can observe how timer value changes with respect to this equation [3.7].

Figure 3.7: Scatter 2D 1



On legend there are descriptions of curves with:

- "Min" - minimum time for every X coordinate
- "Max" - maximum time for every X coordinate
- "Avg" - average value along each X coordinate

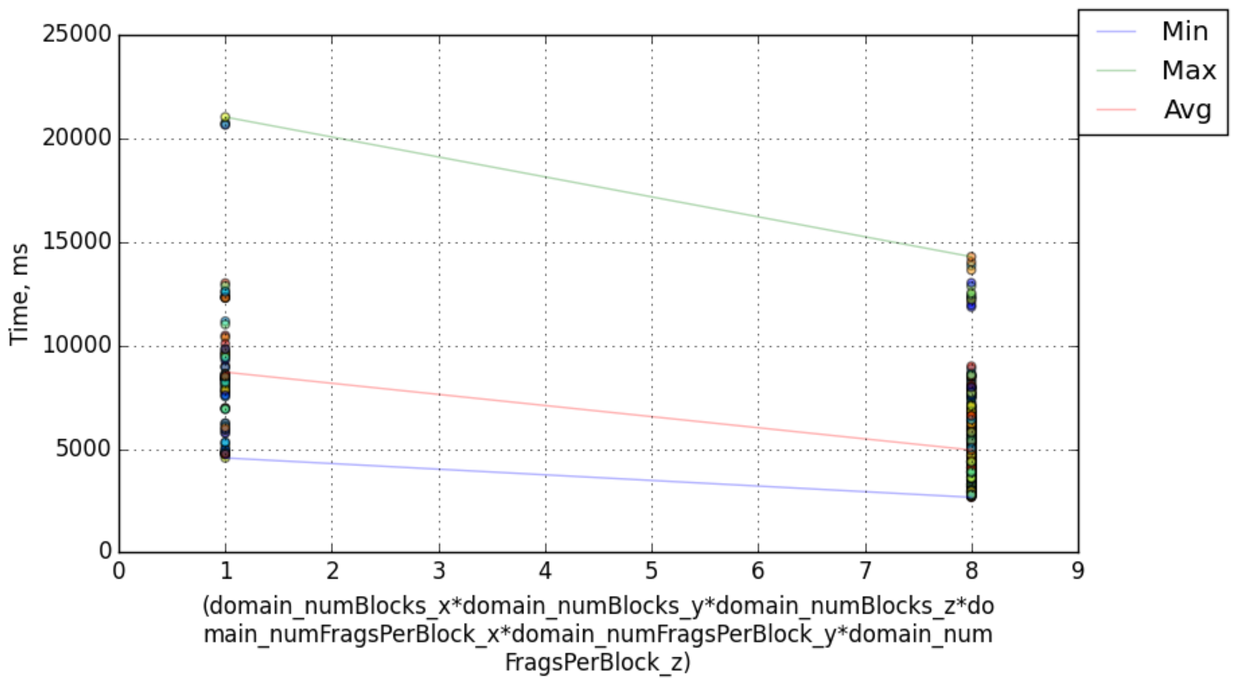
If user wants to observe strong scaling, they should specify (in project file) $strong\_scale = X$, where X is equal, for example, 256, and specify following equation:

$$domain\_numBlocks\_x * domain\_numBlocks\_y * domain\_numBlocks\_z *$$
$$domain\_numFragsPerBlock\_x * domain\_numFragsPerBlock\_y$$
$$*domain\_numFragsPerBlock\_z$$

.

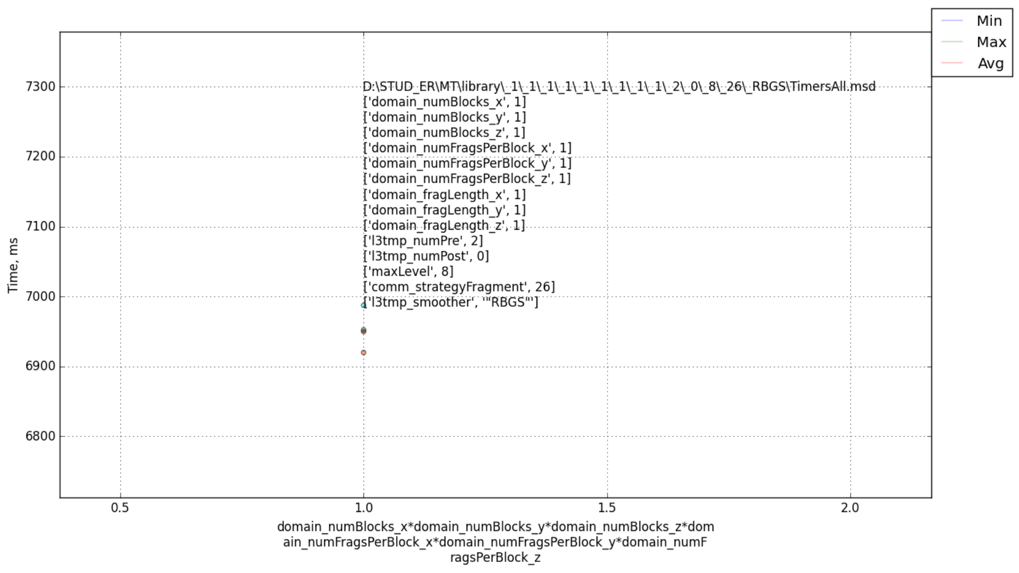
Plot of such setup look is shown on [3.8].

Figure 3.8: Scatter 2D strong scale



In 2D scatter plot there is an option to navigate by zooming and translating. One can navigate to specific unit and click on it in order to observe more information, e.g. absolute path to this profile and genes, used to generate measured unit [3.9].

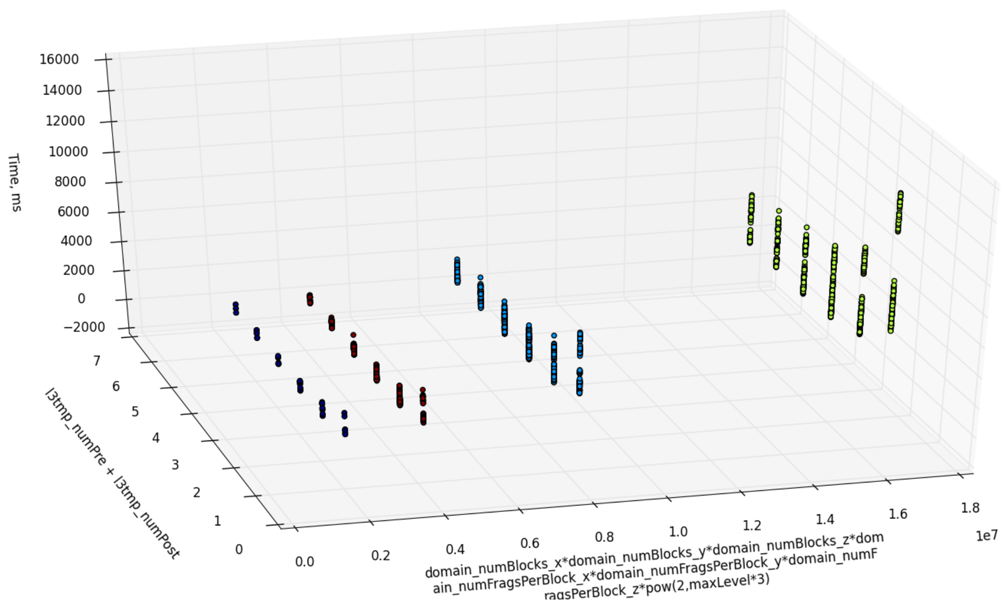
Figure 3.9: Scatter 2D on click



```
D:\STUD_ER\MT\library\_1\_1\_1\_1\_1\_1\_1\_1\_1\_2\_0\_8\_26\_RBGS\TimersAll.msd
['domain_numBlocks_x', 1]
['domain_numBlocks_y', 1]
['domain_numBlocks_z', 1]
['domain_numFragsPerBlock_x', 1]
['domain_numFragsPerBlock_y', 1]
['domain_numFragsPerBlock_z', 1]
['domain_fragLength_x', 1]
['domain_fragLength_y', 1]
['domain_fragLength_z', 1]
['l3tmp_numPre', 2]
['l3tmp_numPost', 0]
['maxLevel', 8]
['comm_strategyFragment', 26]
['l3tmp_smoother', '"RBGS"']
```

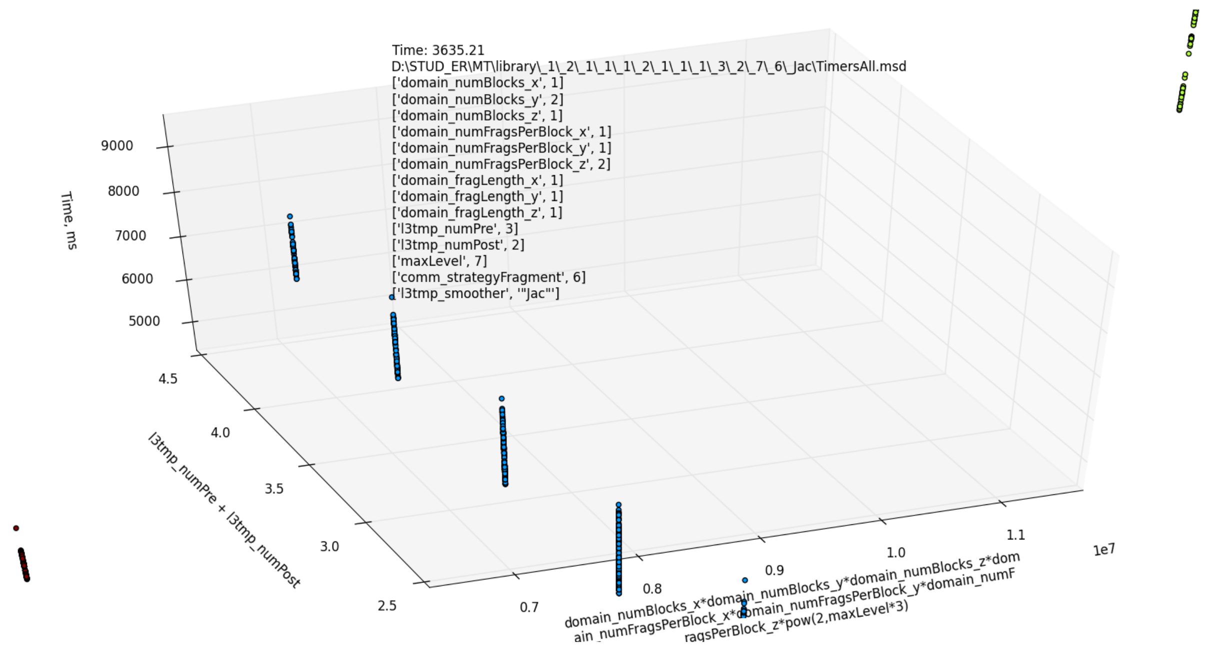
Second option can be used to plot values in 3D. In this case equations for both X and Y axes must be set. This can be used, for example, in case the user wants to observe the impact of the amount of smoothing steps on different problem sizes with different configurations (see [3.10]). The drop-down menu allows to choose option to aggregate or filter units on one X-Y coordinate.

Figure 3.10: Scatter 3D



Clicking on one unit user can receive detailed information in the same way as for 2D case, plus the actual z-value [3.11].

Figure 3.11: Scatter 3D on click



```
Time: 3635.21
D:\STUD_ER\MT\library\_1\_2\_1\_1\_1\_2\_1\_1\_1\_3\_2\_7\_6\_Jac\TimersAll.msd
['domain_numBlocks_x', 1]
['domain_numBlocks_y', 2]
['domain_numBlocks_z', 1]
['domain_numFragsPerBlock_x', 1]
['domain_numFragsPerBlock_y', 1]
['domain_numFragsPerBlock_z', 2]
['domain_fragLength_x', 1]
['domain_fragLength_y', 1]
['domain_fragLength_z', 1]
['l3tmp_numPre', 3]
['l3tmp_numPost', 2]
['maxLevel', 7]
['comm_strategyFragment', 6]
['l3tmp_smoother', '"Jac"']
```

Time, ms
9000
8000
7000
6000
5000
4.5
4.0
3.5
3.0
2.5
l3tmp_numPre + l3tmp_numPost
0.7
0.8
0.9
1.0
1.1
1e7
domain_numBlocks_x*domain_numBlocks_y*domain_numBlocks_z*domain_numFragsPerBlock_x*domain_numFragsPerBlock_y*domain_numFragsPerBlock_z*pow(2,maxLevel*3)

## 3.5 Summary

In this chapter we presented and described the process of automatic batch code generation, compilation and execution[2]. Together with an option to visualize data this allows to implement the last stage of this work, automated performance optimization.

[2]and profiling, respectively

# Chapter 4

# Automated performance optimization

*"In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selection amongst them for the purposes of a Calculating Engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation."*

— Ada Lovelace, *Sketch of the Analytical Engine Invented by Charles Babbage*

The final task of this thesis is automated program optimization. This implies coming up with an optimization technique. We have chosen genetic algorithm to be such technique. It does not guarantee finding an optimal solution, but can help find a good one. Similarly, we can not prove it being very effective for the particular problem.[1].

## 4.1 Introduction

Genetic Algorithms[2] (GAs) are stochastic search methods that mimic process of natural evolution. GA incorporate such operations as selection, mutation, and crossover. This helps to explore search space more intelligently than by using random search. Yet being random, genetic algorithm directs search into regions with better fitness. The search space for GA can be continuous, discrete or mixed. In this thesis search space is set to be discrete. The convergence properties of discrete GA are among the main concepts being explored in this chapter.

## 4.2 Description of Genetic Algorithm

Genetic algorithm can be represented as follows:

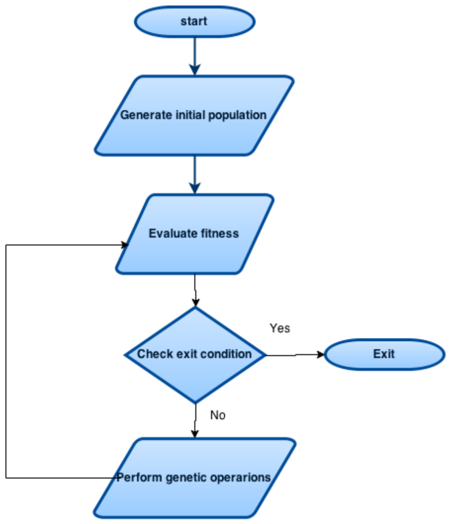
1. Generate initial population randomly.
2. Evaluate fitness.
3. If exit condition is fulfilled then exit.
4. Generate new population using genetic operations - selection, mutation, crossover.
5. Go to 2.

[1]There is a Holland's schema theorem[19], that is widely used as foundation to answer positively the question "Does genetics works?". This theorem is criticized for being not able to predict behaviour of GA over multiple generations, but it can still be used[20]

[2]Idea introduced by I. Rechenberg [21], algorithm introduced and investigated by John Holland [22]

In the graphical form it is presented on figure [4.1].

Figure 4.1: Genetic algorithm description



Initial population is generated randomly. Then fitness evaluation is performed. In scope of this thesis, fitness is time, measured by one of the timers. Here the execution time is subject of optimization. In order to define the source of fitness value we add an additional setting to the project file:

Listing 4.1: Genetics fitness parameter

```
...
timer_to_use = 'stopWatch'
...
```

As an example, we have set a "stopWatch" timer, but it could be any other measured timer. This setting is also used in Processing tool to set initial value of [3.6].

Also, we need a strategy in order to define fitness value, since there is usually more than 1 thread. This option is set by another variable:

Listing 4.2: Genetics processing strategy

```
...
processing_strategy = "avg"
...
```

This value can be any of ['min', 'max', 'avg', 'sum'].

Generally, genetic algorithm is running with some population size and for some (fixed) number of generations:

Listing 4.3: Population settings 1

```
...
population_size = 50
max_generation_count  = 20
...
```

Also, we need to specify mutation chance:

Listing 4.4: Population settings 2

```
...
mutation_chance = 100
...
```

This value means that there is one out of ’mutation_chance’ chance to mutate. If unit was selected to mutate, there is a 50% chance for each gene to mutate into random value ( from values, set in corresponding gene definitions in project file).

## Selection

Selection is the process of selecting units for reproduction to produce next generation of units. There are number of selection methods, like rank selection, truncation selection, roulette wheel selection and so on. In this thesis roulette wheel selection is investigated.

Roulette wheel selection uses fitness values of all units in population and assigns probability for each to be selected as parent by formula:

$$p_i = \frac{f_i}{\sum\limits_{j=1}^{n} f_j}$$

However, this formula will select the ones with greater fitness (time). And since we want the opposite result (assign higher probability to units with lower fitness), formula should be rewritten as follows:

$$p_i = \frac{\frac{1}{f_i}}{\sum\limits_{j=1}^{n} \frac{1}{f_j}}$$

Afterwards we can generate new unit using crossover.

## Crossover

There are numerous ways to perform crossover[3]

1. One-point crossover
2. N-point crossover
3. Uniform and half-uniform crossover
4. Cut and slice

For one-point crossover random point on parent genome is selected and then all genes beyond this point are swapped between parents. One of them can be selected as new unit.

N-point crossover is similar to one-point with the only difference that N points are selected and swap happens for even (or odd) chunks of genome.

In case of uniform crossover all genes are swapped between parents with 50% (typically) probability. In contrast half-uniform crossover at first evaluates amount of different genes and

[3]Some authors [23] show by experiment that uniform crossover has certain advantages.

then exactly half of them are swapped.

In cut and slice approach every parent has its own slice point and this can produce new gene with different length. This is not possible for the optimisation problem considered in this thesis.

In this thesis one-point and uniform crossover are investigated.

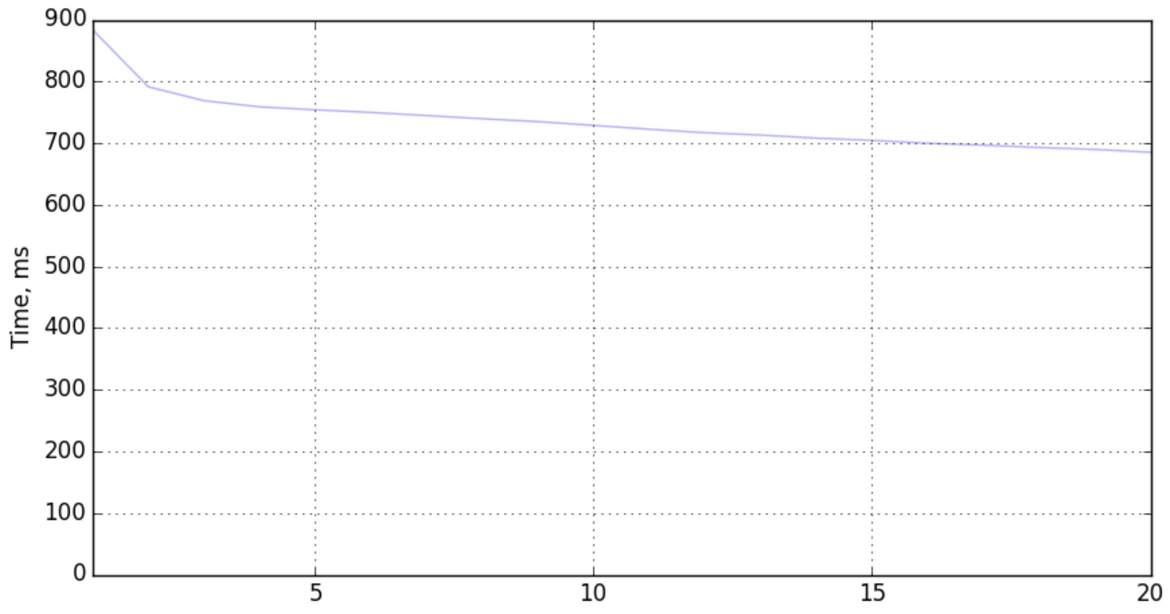
### Elitism

Elitism is the technique that helps to implicitly transit (several) best units from previous generation to a new one. This allows the best units to survive to the end of run.[4] It will be used later.

## 4.3 First version of GA

The very first implementation of GA was simple enough. The uniform crossover was used with elitism. This version did not use any kind of library, so units were generated into a folder with current generation. It takes a long time to GCR[5] every unit, one generation could take up to 20 minutes to be explored. Yet, the results looked good(average mean time for 1000 runs)[6] [4.2].

Figure 4.2: First version GA average convergence



We can clearly see convergence here. It is left to decide how "good"[7] it is.

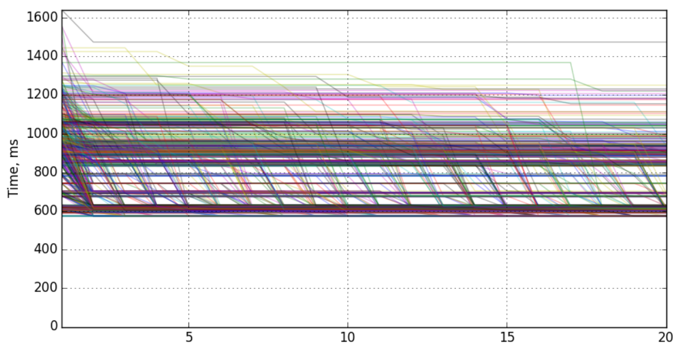
[4]This also prevents population from extinguishing.

[5]Generate-Compile-Run

[6]Actually, first revision would take 9 month to run for 1000 GA runs with 20 generations. This plot is provided by an updated version of script with library support

[7]In the sense of total execution cost

Figure 4.3: First version GA full convergence



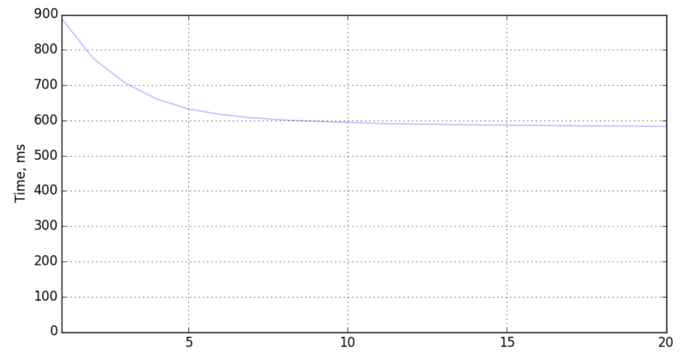
On the plot [4.3] we can observe clustering around certain values. These values are local optimums of search space.

## 4.4 Comparison with random selection

In order to investigate whether selection really improves convergence, another algorithm without actual selection (just random selection of parents) was implemented.[8]

Results for this setup were quite unexpected[9] [4.4].

Figure 4.4: Random parent GA convergence

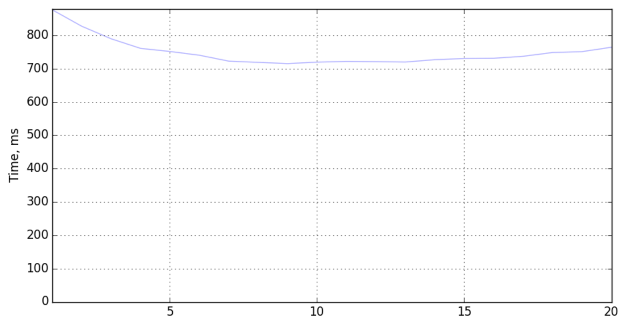


[8]The idea behind it is that if random selection produces same or better result than roulette wheel selection, then GA does not work for current settings.

[9]These results were received because we used elitism.

In order to investigate why it happens, elitism was disabled [4.5].

Figure 4.5: Random parent GA convergence no elitism



It turns out that algorithm has a high chance to hit "good" units during execution, and, therefore, in the second test run, the number of unique units in one genetic run was output. The resulted value was 8 times greater than the one gotten from genetics with roulette wheel selection.

## 4.5 Improved restriction for genetics

Since the GCR routine is much more expensive in time than genetic algorithm execution, we decided to stop genetics not after certain amount of generations (or till all units converge to one), but rather after certain amount of unique units were checked. This restriction can help compare the actual convergence of different types of genetics. That is because with big enough number of unique samples the chance to hit a "good" unit can increase faster as compared to the result provided by optimization technique.

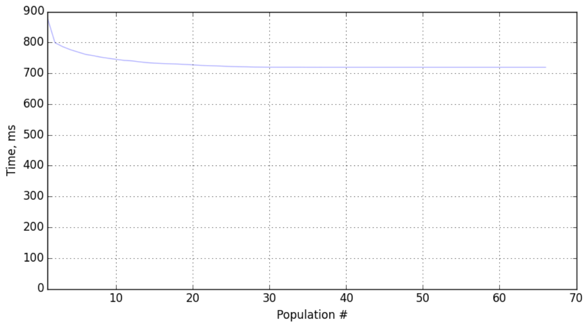
Here "max_generation_count" is removed and "max_unique_units" variable is introduced instead:

Listing 4.5: Population settings 3

```
...
max_unique_units = 50
...
```

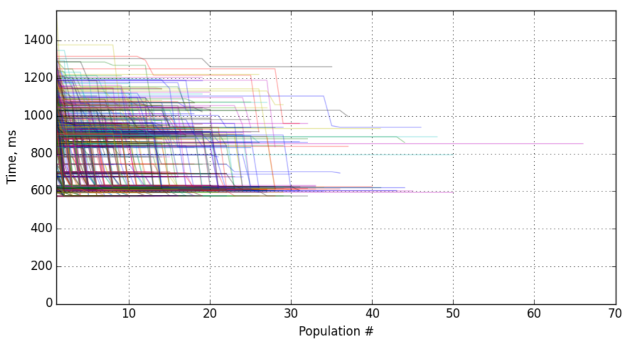
Also, elitism is used [4.6].

Figure 4.6: GA convergence with uniform crossover and restriction



Full convergence plot is presented on [4.7][10].

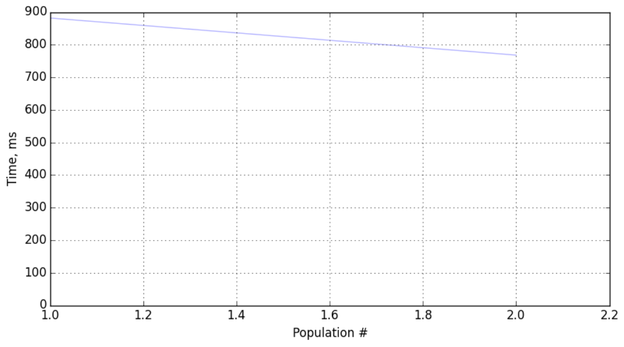
Figure 4.7: GA with uniform crossover and restriction, full convergence



These results can be compared with the results of random selection algorithm. This will allow to make conclusion about GA performance.

Because the random selection without elitism showed the worst possible convergence, elitism was used [4.8][11].

Figure 4.8: GA with random selection, elitism and restriction, 10k runs



So the conclusion about GA performance in the setup with roulette wheel selection, uniform crossover, elitism and unique unit count limiter is as follows:

1. The lower bound of GA performance is the same as that for random selection.
2. Unique unit count limiter guarantees the upper bound for the total runtime, including GCR routine.
3. It was shown experimentally that GA can actually help find "good" solution.

However, optimization of current[12] problem using GA requires more experiments and investigation.

[10]Because there is no generation limit, "X" axis for this setting goes up to 66 generations.
[11]Even being limited, for this plot average unique unit count is 76.
[12]Optimisation of code provided by exastensils metacompiler

## 4.6 Strong scalability and genetics

Up to this point genetics was tested on a weak scaled problem, and it actually converges to setup with fewer number of threads. But this is not the actual purpose of the performance optimization.

For now it was shown that GA can actually find "good" solution at reasonable cost and time. Now the question is whether the same algorithm can provide optimized program in case of a strong scaled problem. For this purpose the setting named "strong_scale" should be used in project file. It is set to 256.

Since this restriction forces the script to generate problems with constant problem size, it is possible to investigate strong scale. Due to fact that the total amount of unique units is less than for a weak scaled setup, the number of unique units per genetic run was decreased to 20, "population_size" to 30.

Final project file for this is presented below:

Listing 4.6: Projects file for strong scale genetics

```
cmp_path = '/cygdrive/d/STUD_ER/MT/CodeGenerator/'

range_genes_def =[
["domain_numBlocks_x",1,1],
["domain_numBlocks_y",1,2],
["domain_numBlocks_z",1,2],
["domain_numFragsPerBlock_x",1,2],
["domain_numFragsPerBlock_y",1,2],
["domain_numFragsPerBlock_z",1,2],
["domain_fragLength_x",1,1],
["domain_fragLength_y",1,1],
["domain_fragLength_z",1,1],
["l3tmp_numPre",0,4],
["l3tmp_numPost",0,4],
["maxLevel", 7, 1]#ignored actually.
]

list_genes_def = [
["comm_strategyFragment",[6,26]],
["l3tmp_smoother", ["\"GS\"","\"Jac\"","\"RBGS\""] ]
]


const_genes = [
["targetCompiler","\"GCC\""],
["simd_instructionSet","\"SSE3\""],
["dimensionality",3],
["domain_summarizeBlocks", "false"],
["omp_parallelizeLoopOverDimensions", "false"],
["l3tmp_genFunctionBC", "false"],#only for 2D
["l3tmp_genAdvancedTimers","true"],\
["advTimer_timerType","\"QPC\""]
]

#basic restrictions
generate_predifined = False
num_omp_per_mpi = [1,8]
num_mpi = [1,8]
num_total_threads = [1,8]

run_while_generating_or_genetics = True
out_path = '../out_genetics_strong/'
lib_path = '../library/'
```

```
#advanced restrictions
strong_scale = 256

#file name for timers
measured_timers_fname = "TimersAll.msd"

#genetics settings
population_size = 30
max_unique_units = 20
timer_to_use = 'stopWatch'
mutation_chance = 100
processing_strategy = "avg"
use_elitism = True
```

Script call for genetics looks like this:
"Python3 PATH_TO_SCRIPTS/GenerateGenetics.py -f project.gp -r X"

Here "-r"[13] option can be set if user wants to run GA X times. If this option is not specified only one GA run will be performed.

It will output intermediate data into folder, specified by "out_path" variable. This intermediate data is:

1. Subfolders with name "genetic_generation_X", where X is generation number. In each of these subfolders the best unit is saved as corresponding Knowledge file with name "Best: FITNESS" where "FITNESS" is the fitness of this unit.
2. Convergence file (*.cnv). File's name is "Convergence" if there was one execution of GA, or "Convergence_I" if -r key was specified . "I" is the repeat index.

Example Convergence file is presented below:

Listing 4.7: Example Convergence file

```
21
4241.24;3541.21;3069.3099999999995;2905.51;2905.51; ...
```

First line here denotes the number of unique units for this genetic run. Next line presents the sequence of best finesses, one per generation.

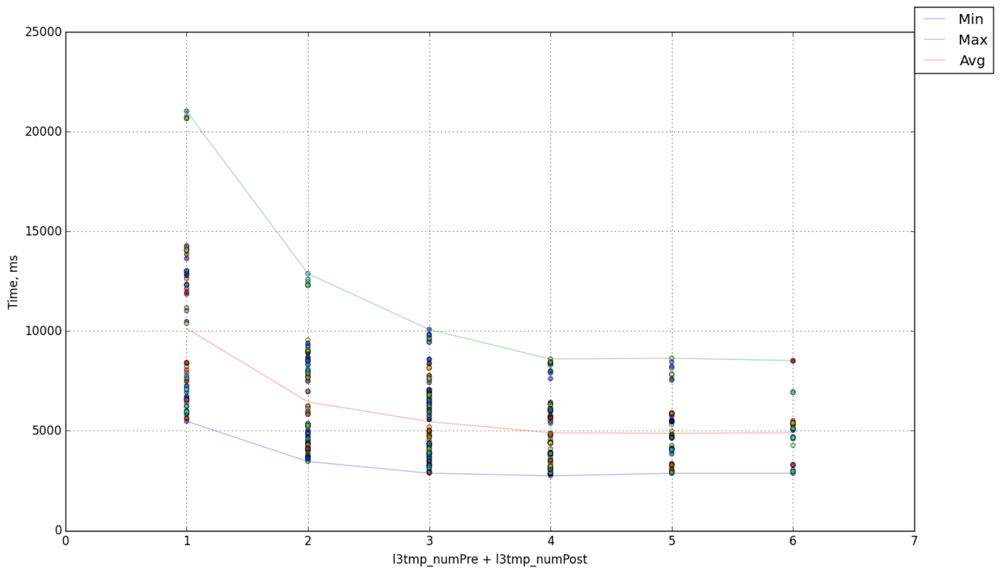
[13]r stands for "repeat"

## 4.7 GA performance for strong-scaled problem

The setup presented in listing [4.6] produces 450 possible combinations, so here we are actually sampling 4.4(4)% of them.

To give the idea about fitness distribution of these 450 units, example plot of them is presented [4.9].

Figure 4.9: Strong scaled distribution w.r.t. smooth step



The script was executed for 1000 iterations for two GAs: one with one-point crossover, another with uniform.

The resulting convergence plots look similar [4.10], [4.11].

Figure 4.10: Strong scaled uniform GA convergence

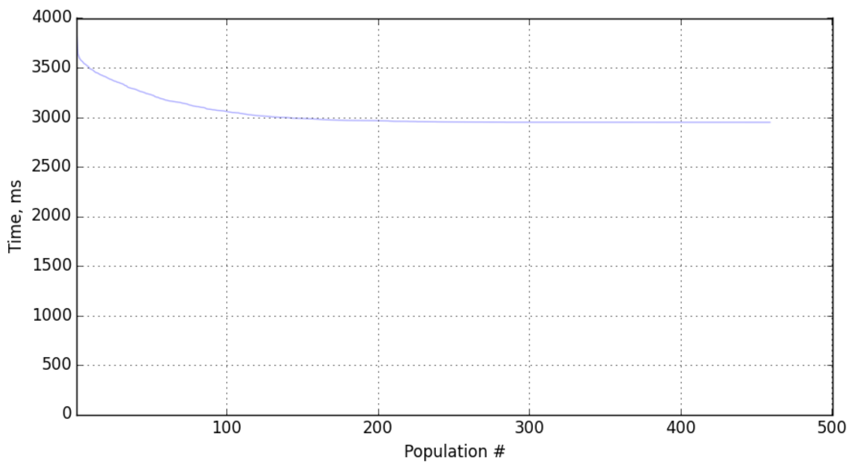
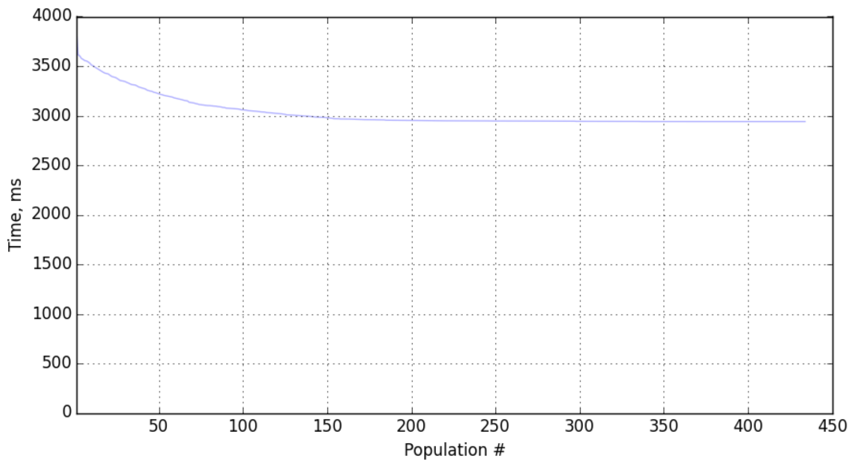


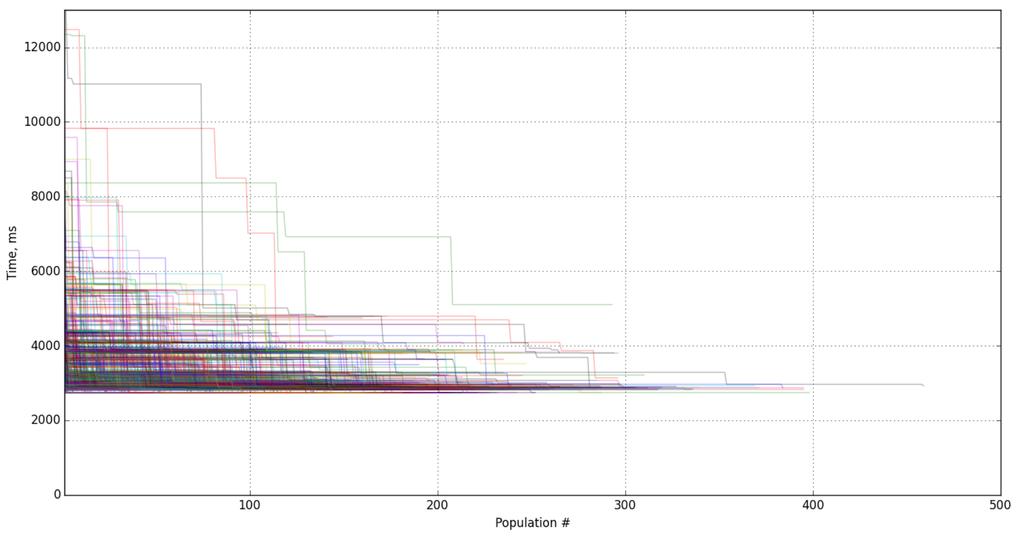
Figure 4.11: Strong scaled one-point GA convergence



Both of them converge to units with fitness close to 2950.

Full convergence plot for uniform GA is presented here on plot [4.12].

Figure 4.12: Strong scaled uniform GA full convergence

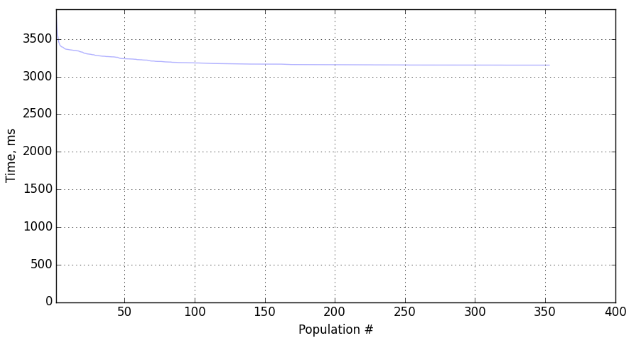


All tested runs of GA are clustering below fitness value of 3000, which is a good result[14] of search algorithm w.r.t. distribution presented in [4.9].

It would also be a good idea to compare these results with random selection GA with elitism. Resulting plot is presented on [4.13]. Clearly, convergence is much worse here, so we can conclude that GA with uniform or one-point crossover does actually help to find a "good" solution.

[14]taking into account that only 4.5% of all possible configurations were tested on every run

Figure 4.13: Strong scaled random GA full convergence



So the answer to all questions about performance mentioned in this chapter is yes. GA is effective for both strong and weak scaled problems, and it provides a reliable result.

## 4.8 Summary

In this chapter genetic algorithm was described and implemented. Being a final step of pipeline described in this work, it uses both performance measurement and automatic batch processing, described in Chapter 3. For problem and setup, defined in this chapter GA provides good results and theoretically can be used for automatic performance optimization.

However, more research is necessary in order to find better GA setups for this problem. It is possible that Adaptive GA or Parallel AGA will produce better results within reasonable runtime.

# Chapter 5

# Results

In this chapter we present the results of thesis.

The aim of this thesis has been to develop a toolchain for automatically performing performance measurement and measured data analysis of highly parallel programs.
In Chapter 1 we investigated different performance measurement approaches and formulated the requirements to performance measurement tool.

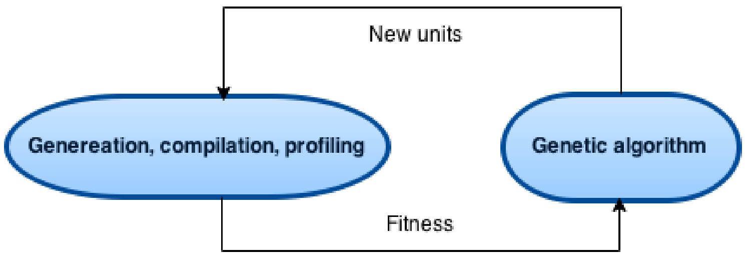
Second Chapter defines important concepts regarding performance measurement, investigates the precision of different time measurement functions. Further more it defined the Stopwatch class capable of both time measurement and call stack tracking on different platforms. The profile format was described, the visualization script (for profile and call stack) was presented.

In third Chapter, the process of automatic code generation, compilation and execution was presented and implemented in Python script. Also, we presented the project file format which encapsulated all necessary data for Python script. This script is the core of all automated actions, including code generation, compilation and profiling. Additional visualization tool for this project file was shown.

Finally in Chapter 4 is describing genetic algorithm as coupling method between code generation and code profiling, resulting in the full cycle [5.1] of performance optimisation. Important concepts were reviewed regarding discrete genetic algorithm and its convergence properties.

Developed tool chain allows to automatically perform performance optimisation and visualise all stages of this process.

Figure 5.1: Full cycle of performance optimisation



# Chapter 6

# Suggestions for Future Work

One of the most important research directions for performance measurement is integration with likwid. In the future version of likwid it will be possible to read measured time from inside an application. This will help to make Stopwatch portable without necessity to support heavy Scala stopwatch generator.

Second important research direction is to provide further exploration of discrete and adaptive discrete genetic algorithms, and its applicability for automatic performance optimization.

Third research direction can be seen in creating fully parallel optimization pipeline. The idea behind this is since we rely on weak scaling, we can perform several profiling run at the same time on big cluster. Suppose cluster has 128 cores and application is limited to use no more than 32, then it is a good idea to have 4 instances run at the same time. For now it is not possible within presented toolchain.

# Bibliography

[1] http://www.infosun.fim.uni-passau.de/publications/docs/KoestlerHabil2013.pdf.

[2] http://www.exastencils.org/.

[3] https://www.python.org/.

[4] http://fenicsproject.org/.

[5] http://www.scipy.org/index.html.

[6] https://svn.boost.org/trac/boost/ticket/7719.

[7] https://connect.microsoft.com/VisualStudio/feedback/details/719443/.

[8] http://msdn.microsoft.com/en-us/library/windows/desktop/ms644904

[9] http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf.

[10] https://code.google.com/p/likwid/.

[11] http://www.cs.uoregon.edu/research/tau/home.php.

[12] https://www.gnu.org/software/libc/manual/html_node/PowerPC.html.

[13] https://tools.ietf.org/html/rfc4180.

[14] http://matplotlib.org/.

[15] https://www.incredibuild.com/webhelp/monitor.html.

[16] Christian Iwainsky, Jan-Patrick Lehr, and Christian Bischof. Compiler supported sampling through minimalistic instrumentation. In *Parallel Processing Workshops (ICPPW)*. 43st International Conference, 2014. to appear.

[17] Christian Schmitt, Sebastian Kuckuk, Frank Hannig, Harald Köstler, and Jürgen Teich. Exaslang: A domain-specific language for highly scalable multigrid solvers. In *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '14, pages 42–51, Piscataway, NJ, USA, 2014. IEEE Press.

[18] http://www.scala-lang.org/.

[19] Clayton L Bridges and David E. Goldberg. An analysis of reproduction and crossover in a binary-coded genetic algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*, pages 9–13, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc.

[20] K.S. Tang, T.M. Chan, R.J. Yin, and K.F. Man. *Multiobjective Optimization Methodology: A Jumping Gene Approach (Industrial Electronics)*. CRC Press, 2012.

[21] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog, 1973.

[22] J.H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975.

[23] William Spears. On the virtues of parameterized uniform crossover, 1991.

# Chapter 7

# Additional

## 7.1 Known problems

The most important problem is that Stopwatch profile, being printed in different translation unit than one were timers are created, will result in empty profile.

Second problem is that matplotlib library can be inaccessible on linux machines if the backend is not set properly.

## 7.2 Manual for scripts

### GenerateGenetics.py

"GenerateGenetics.py" is the core script, capable of both generating all possible units and performing genetic algorithm.

#### Command line arguments

There are different possible combination of command line arguments:

"python3 PATH_TO_SCRIPTS/GenerateGenetics.py -f project.gp -g"
Will generate all possible programs w.r.t. project file.

"python3 PATH_TO_SCRIPTS/GenerateGenetics.py -f project.gp"
Will perform genetic algorithm w.r.t. project file.

"python3 PATH_TO_SCRIPTS/GenerateGenetics.py -f project.gp -r X"
Will perform genetic algorithm w.r.t. project file, repeating this process X times.

### ConvergancePlotter.py

Is capable of plotting convergence files produced by "GenerateGenetics.py". Has no command line arguments arguments.

## TheTool.py

Incorporates all plotting functions, uses project file as source to build interface and load measured data. Has no command line arguments.