UNIVERSITY OF BASEL - DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

Visualisation of Scheduling Algorithms

Aram Yesildeniz
June 22, 2016

The target of this project is the vizualisation of parallel scheduling algorithms. This will require parsing of the output of existing scheduling simulators and development of a tool for transforming this information into a trace written in the open trace format (OTF2). This trace can subsequently be visualised by a trace visualiser (e.g. Vampir).

CONTENTS

1 INTRODUCTION

Due to an enormous technical progress in the last years, *High-Performance Computing (HPC)* has become one of the most important and actively investigated disciplines in computer science. The main objects of research are distributed systems like clusters, supercomputers, grids or clouds. As one would expect, evaluations, performing measurements and corrections of these complex, large and dynamic structures are as essential as complicated.

Performance optimisation is one goal. Making predictions about the performance of such applications requires experimentation. There are three ways of executing experiments: Real applications in real environments (in vivo), real applications on synthetic platforms (in vitro) and as a simulation with prototypes of applications on system's models (in silico).

A main principle in science is the reproducibility and validation of experiments. Thus, there is a need for standard tools and techniques with which experiments can be imitated and compared. In regard to the experimentation as simulation, one of the standard tools is *SimGrid*.

1.1 SIMGRID

SimGrid is a versatile simulator of distributed applications. As a scientific instrument, SimGrid provides many-sided functionalities and possibilities of conducting simulations. With respect to the scope of this project, only the specific output files, produced by SimGrid within another academic venture, are of relevance.

The output file is basically a trace file in which generated information during the simulation's execution is collected (*Listing 1*). The heart of the file is a table containing data about the distribution and disposal of *chunks* (column "chunkID"). Each chunk contains an amount of *tasks* ("#Tasks") and has a starting time ("startTime") and an end time ("endTime"). The chunks are grouped together within *batches* ("batchID"). A batch is a collection of chunks which will be processed by *hosts* ("hostID").[1] The chunks are the unique elements considering identification (in a special case, where one batch includes one chunk, batches are likewise unique). In view of the table it becomes clear that one batch is not processed uninterrupted on one host, but split up into several pieces which are spread over many hosts in a parallel manner.

The task of the project is the visualisation of the data introduced above. Visualising abstract knowledge is a common and admired way of showing information more clearly and cohesively. It makes complex data more accessible, understandable and usable. Facing the problem of visualising numbers, there are different tools available. A well known and widely used one, as well within this project, is *Vampir*. Due to the format of the given SimGrid output, Vampir can not read the file as it stands. As a consequence, a conversion of the SimGrid output file to a format readable by Vampir, is required. Such a format is the *Open Trace Format 2 (OTF2)*.

[1] Where the term "host" is referring to a physical processor, in this context the correct term should be "worker", which describes a logical processor. Since in this example one worker is assigned to one host, they are equivalent.

```
platform file: GP1000_80hosts_star_platform_1KFLOPS.xml
bandwidth = 2.5E3 Byte/s
latency = 1E-6 s
pe power = 1E3 FLOPS
deployment: deployments/8worker1master_deployment.xml
#tasks=128
Communication size of tasks=0.000000
DLS Technique=Fixed Size Chunking
Task execution time distribution: uniform distr. over the interval
    (mu-sqrt(3)*sigma, (mu+sqrt(3)*sigma))
h=0.500000
mu=1.000000
sigma=0.200000

hostID  batchID chunkID #Tasks  startTime       endTime
2       0       0       12      0.000459        0.012430
4       2       2       12      0.001377        0.013089
3       1       1       12      0.000918        0.013466
5       3       3       12      0.001836        0.013990
7       5       5       12      0.002754        0.014473
8       6       6       12      0.003213        0.015927
6       4       4       12      0.002295        0.016169
9       7       7       12      0.003672        0.016187
3       10      10      8       0.014007        0.021638
2       8       8       12      0.012889        0.024683
4       9       9       12      0.013548        0.025152

batch counter= 11       chunk counter= 11       task counter=128

Simulated execution time 0.0256113 seconds

sum worker task exec time: 0.130235
```

Listing 1: SimGrid Output File

1.2 OTF2

> The Open Trace Format 2 is a highly scalable, memory efficient event trace data format plus support library. It will become the new standard trace format for Scalasca, Vampir, and Tau and is open for other tools.[2]

Regarding performance analysis of applications, several approaches exist, like the dynamic program analysis methods *profiling* and *event tracing*. While profiling measures and collects different performance metrics, more data and details can be fetched with event tracing, which records all events, such as entering and leaving of functions or sending and receiving of messages. OTF2 is a format for trace-based analysis tools.

[2] http://www.vi-hps.org/projects/score-p/ [12.06.2016]

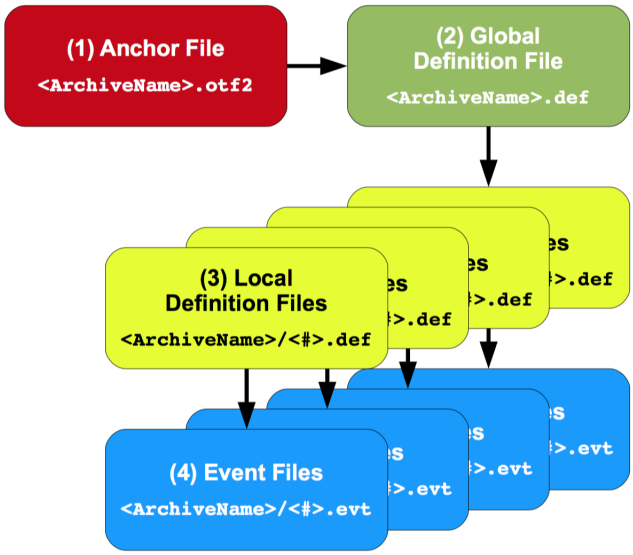

Figure 1.1: OTF2 File System Layout[5]

Within the different trace file formats that exist, OTF2 arose as a successor and combination of the Open Trace Format (OTF), used by Vampir, and EPILOG, the standard format of Scalasca. OTF2 is closed integrated in the trace recording system Score-P. A major benefit is the compatibility of the new format with the tools Vampir and Scalasca, a fact what does not apply for the predecessors OTF and EPILOG. Consequently the user can analyse an application with both analysis tools in one single measurement run.

A second key feature is the scalability of the supporting numbers of processes and threads and time, which is in a HPC context of particular significance. The scalability is made possible with specific compression techniques. Other improvements and beneficials are a read/write support library with an API[3] and a modular design that makes the library easily accessible and extendable.[4]

The OTF2 file format is understood as an archive of four different file types (see *Figure 1.1*): The *Anchor File* contains all meta data relevant for the archives management, the *Global Definition File* definitions which are valid and equal for all locations, the *Local Definition Files* and *Event Files* contain specific definitions for the locations and corresponding event data respectively.

In regard of this project, the OTF2 terminology is linked to the SimGrid elements as follows: a *location* in OTF2 is as a *worker* (or host) in the SimGrid file. In Vampir, a location is one entry on the left-sided "Process Bar". A *region* in OTF2 corresponds to a *chunk* in SimGrid.

[3] https://silc.zih.tu-dresden.de/otf2-current/html/ [19.06.2016]
[4] see: Eschweiler, D. et al. (2012): Open Trace Format 2. The Next Generation of Scalable Trace Formats and Support Libraries. [Eschweiler (2012)]
[5] Eschweiler (2012).

The region is mapped to the chunk structure due to his uniqueness, thus each region/chunk is distinctive inscribable. The *events*, which are mapped to the regions, featuring entry and leaving points that are the *start* and *end* times of the chunks.

The OTF2 library is organised in four layers. Considering the scope of this project, layer two, the record representation, is appreciable, meaning it consists of components through which trace data can be stored and accessed.[6] How trace data can be read out of the SimGrid output file and be written to an OTF2 archive will be discussed below.

2 HIPPIE2

2.1 OVERVIEW

Given the introduction to the project and the problem definition, the task can be subdivided into two jobs:

- Read SimGrid output file and store data
- Write OTF2 file

The writing task has to take the advantages of the OTF2 API. The numerical values of the SimGrid file, the quantities of workers, batches, chunks and tasks and their aggregation, have to be translated and written to corresponding OTF2 components. As previously reported, the workers are written as locations, the chunks and timestamps as regions and events. More details about the writing task is given in subchapter *2.3.1 hippie2.c*.

The reading process is rather simple and does not require the usage of the OTF2 API. In subchapter *2.3.2 parser.c*, the functions that were implemented to read the SimGrid output file and to store the data in a way it can easily be reused, will be discussed.

As a matter of fact, the chunks are organised within batches. In terms of OTF2, where a chunk is a region, a batch would be a group of regions. During the projects progress it turned out that grouping regions together is not a straightforward task as one might assume. Therefore, the lines of code needed for solving this problem are examined in subchapter *2.3.3 groupwriter.c*.

2.2 USAGE

After a successful installation of the OTF2 library, the source files of the hippie2 project can be compiled: The OTF2 headers must be included and the object files have to be linked against the library:

[6] For more details see Eschweiler (2012).

```
gcc -std=c99 -I/opt/otf2/include -c parser.c -o parser.o
gcc -std=c99 -I/opt/otf2/include -c groupwriter.c -o groupwriter.o
gcc -std=c99 -I/opt/otf2/include -c hippie2.c -o hippie2.o

gcc parser.o groupwriter.o hippie2.o -L/opt/otf2/lib -Wl,-rpath
    -Wl,/opt/otf2/lib -lotf2 -lm -o hippie2
```

Listing 2: Compiling in Terminal

Alternatively, the correct options for a successful compiling can be fetched with the OTF2 tools (see "otf2-config –help" for more details). Before the OTF2 tools can be used, their path has to be appended to the PATH environment variable:

```
export PATH=$PATH:/opt/otf2/bin/
```

Listing 3: Set PATH Variable

Eventually the compiling task can be performed using the Makefile by using the keyword "make".

The program runs as follows:

```
./hippie2 [OPTION] FILE
```

Listing 4: Hippie2 Usage

Additionaly, the program can handle several options, one at a time:

- -h: show help text
- -w: if in the SimGrid output file "#hosts" is not available but "#workers", the "-w" option is used to read "#workers" instead of "#hosts"
- -p: print the table from the file in terminal

2.3 IMPLEMENTATION

2.3.1 HIPPIE2.C

The most important functions used from the API are these for writing definitions of strings, locations, regions and events. All these definitions are stored in one global definition file in a continuous manner, where the locations and regions point to string definitions, which contain e.g. appropriate names. Events are written for a specified location and are associated with a region definition. The functions are:

- OTF2_GlobalDefWriter_WriteString: Writes a string definition record into the GlobalDefWriter. Global definitions are stored continuously in a file. The strings get referenced directly from locations and regions
- OTF2_EvtWriter_Enter / OTF2_EvtWriter_Leave: Writes an enter/leave record on a specified location and is associated with a region
- OTF2_GlobalDefWriter_WriteLocation: Writes a location definition record into the GlobalDefWriter
- OTF2_GlobalDefWriter_WriteRegion: Writes a region definition record into the GlobalDefWriter

Coming, the prime content from the source file will be explained. The description goes along the lines as they stand in the source code.

First, global variables are declared for storing the values read from the SimGrid output file. Since the definition records are stored in a continuous way, DEFCOUNTER is used as a pointer to the upcoming entry in the global definition file. The start and end times of the chunks are given as decimal digits. OTF2 can not write floats, hence MICRO is used as a factor for shifting the floats to integers. For the time interval to stay the same, the constant is also used as the timer resolution in the clock properties. Note that this is hard-coded and has to be changed if the number of significant digits in the SimGrid output file changes.

```
static unsigned int HOSTS;
static unsigned int BATCHES;
static unsigned int CHUNKS;
static float TIMESPAN; // simulated execution time

static unsigned int DEFCOUNTER = 0; // keeps track of definition records
static unsigned int MICRO = 1000000; // constant for multiplying floats
```

Listing 5: Global Variables

A row from the table is stored in a defined struct called *Chunk*. Each field of the struct represents one column from the table. The Chunk structs are collected in an array called *chunkList*.

```
typedef struct {
        int chunk;
        int host;
        int batch;
        int tasks;
        float startTime;
        float endTime;
} Chunk;

...

Chunk chunkList[CHUNKS];
```

Listing 6: Chunk Struct and chunkList

Next, several preparations like the archive opening and OTF2 internal properties as flush callbacks have to be set.[7]

The chunkList is the main structure which contains all relevant data and which is passed as an argument to the functions that write definitions to the global definition file.

```
// write events: enter and leave event for every chunk
createEvents(archive, chunkList, eventsPerHost);

// create global hosts (worker/process) definitions
createLocations(archive, global_def_writer, eventsPerHost);

// create region definitions (for every chunk one definition)
createRegions(global_def_writer, chunkList);
```

Listing 7: Function Calls with chunkList as Argument

In the event, location and region creation functions, the chunkList is traversed and according to the values, a definition is written to the global definition file.

[7] For more detail see: https://silc.zih.tu-dresden.de/otf2-current/html/group__usage__writing.html [19.06.2016]

```
for (i = 0; i <= HOSTS; i++) {

        ...

        // definition record
        OTF2_GlobalDefWriter_WriteString(global_def_writer, DEFCOUNTER,
            locationName);

        OTF2_GlobalDefWriter_WriteLocation(
                global_def_writer,
                i, // id
                DEFCOUNTER, // name as written in string in global definitions
                    at position "DEFCOUNTER"
                OTF2_LOCATION_TYPE_CPU_THREAD,
                eventsPerHost[i], // number of events per host
                0 // location group
                );

        ...

        DEFCOUNTER++;
}
```

Listing 8: Write Location

As a goal of the project, each chunk has to be labeled unique such as "BA0.CH0.T10". Since a chunk corresponds to a region, this task takes place during the region definition. The specific values are fetched from the chunkList:

```
sprintf(regionName, "%s%d%s%d%s%d", "BA", chunkList[i].batch, ".CH",
    chunkList[i].chunk, ".TA", chunkList[i].tasks);
```

Listing 9: String for Region Name

2.3.2 PARSER.C

The parser file consists of two functions. The getIntNumber/getFloatNumber functions are used for reading out numbers belonging to a given pattern like "#hosts" or "batch counter= ". The found numbers will be returned. The readTable function is used for reading out the table from the file and to store all entries in the arrays which are passed to the function as arguments.

```
int getIntNumber(FILE *readFile, char *pattern);
float getFloatNumber(FILE *readFile, char *pattern);

void readTable(FILE *readFile, int chunks, int hostID[], int batchID[], int
    chunkID[], int tasks[], float startTime[], float endTime[]);


[hippie2.c]

...

if (workerFlag) {
        HOSTS = getIntNumber(readFile, "#workers = ");
} else {
        HOSTS = getIntNumber(readFile, "#hosts = ");
}
BATCHES = getIntNumber(readFile, "batch counter=");
CHUNKS = getIntNumber(readFile, "chunk counter= ");
TIMESPAN = getFloatNumber(readFile, "Simulated execution time");

...
```

Listing 10: Parser Functions

```
void readTable(FILE *readFile, int chunks, int hostID[], int batchID[], int
    chunkID[], int tasks[], float startTime[], float endTime[]) {

        ...

        // read table line by line and save values in arrays
        int i;
        for (i = 0; i < chunks; i++) {
                fgets(line, lineSize, readFile);
                sscanf(line, "%d %d %d %d %f %f", &hostID[i], &batchID[i],
                    &chunkID[i], &tasks[i], &startTime[i], &endTime[i]);
        }

}
```

Listing 11: Store Data from Table

2.3.3 GROUPWRITER.C

Part of the project's goal is the utilisation of colours in an intelligent way to clarify the correlations in the data. As a matter of fact, colouring the batches is one task to achieve. Vampir differs the regions from each other, which makes it possible to handle colours of regions separately. The problem occurs because a region is a chunk and not a batch: A region has to

be a chunk due each chunk is labeled unique ("BA0.CH0.TA10", "BA0.CH1.TA10"). This is only possible, if a chunk is a region.

A batch is a group of chunks, meaning a group of regions. Since there is no option in OTF2 to group regions together (grouping locations is possible though), an alternative way has to be found to solve the batch colouring problem:

(a) One solution may be to declare different functions in the Vampir preferences and assign the regions according to their batch number to a function. Because this has to be done manually by the user, this approach is unsatisfactory.

(b) A restricted, but automated solution is the usage of paradigms. In OTF2, a set of paradigms is available to specify regions. Paradigms are e.g. "OTF2_PARADIGM_USER" or "OTF2_PARADIGM_MPI". Since Vampir differentiates paradigms and colourises them separately, this approach may be used: Each chunk/region is given a certain paradigm according to its batch number. However, the list of paradigms is limited[8], which makes this approach unsatisfiable as well.

(c) Instead of declaring functions manually in Vampir as proposed in approach (a), the declarations can be written in a file (*Listing 12*) which then is read by Vampir. The file has to be named "scorep.fgp" and has to be placed in the same folder as the OTF2 file. Wildcards as the asterisk are allowed. This in the project used solution is still a hidden feature and only available in Vampir 9.

```
BEGIN OPTIONS
        MATCHING_STRATEGY=FIRST
        CASE_SENSITIVITY_FUNCTION_NAME=NO
        CASE_SENSITIVITY_MANGELED_NAME=NO
        CASE_SENSITIVITY_SOURCE_FILE_NAME=NO
END OPTIONS

BEGIN FUNCTION_GROUP Batch_0
        NAME=BA0.*
END FUNCTION_GROUP


BEGIN FUNCTION_GROUP Batch_1
        NAME=BA1.*
END FUNCTION_GROUP

...
```

Listing 12: scorep.fgp

[8] see: https://silc.zih.tu-dresden.de/otf2-current/html/OTF2__GeneralDefinitions_8h.html#aa14d0751354081d258913145a80e79a9 [20.06.2016]

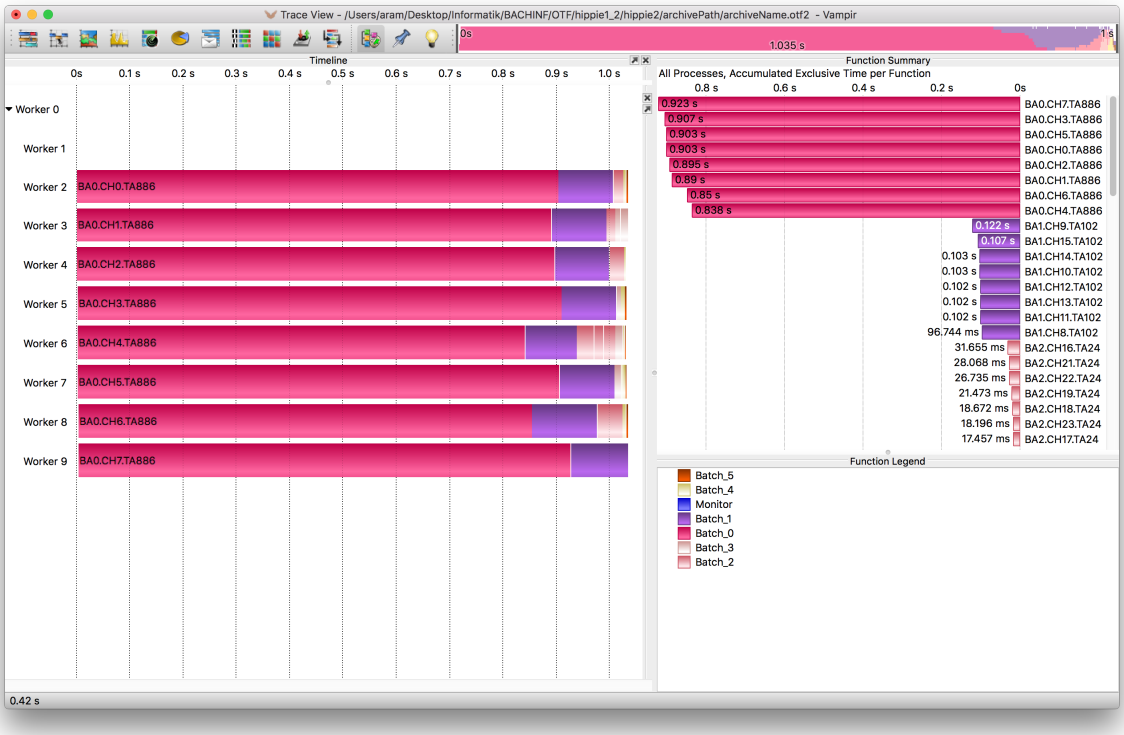

Figure 3.1: Vampir Display

3 Conclusion

The program *hippie2* can read a SimGrid output file and rewrite the information in an OTF2 file, which is displayable with Vampir. This takes place in two major steps: Reading a SimGrid output file and writing an OTF2 file. The source code is divided into three files: In hippie2.c, the main and writing functions occur. Parser.c is responsible for reading specific numbers from the SimGrid output file. Groupwriter.c undertakes the job of grouping chunks together, so that eventually batches are colourable. An example of a Vampir display of a SimGrid output file is given in *Figure 3.1*.

3.1 Main Issues

Mentionable as a main issue was my lack of knowledge, particularly in HPC. Due to the missing expertise, spotting the "big picture" was rather difficult and resulting in a tunnel view work mode.

An other hurdle was the access to information and explanation about OTF2. Besides a

paper[9], the official documentation was helpful.[10] Finally the main sources considered were the examples.[11]

3.2 Further Development

Since the demo version of Vampir is restricted to 16 workers, testing for files which describe more than 16 workers has yet to be done. The program works if and only if the format of the SimGrid output file is as proposed. For further development, a more flexible and generic outline of the program is worthwhile, so that different input formats could be accepted. Finally, imaginable are more options which could reply to upcoming demands, like grouping locations together or colouring specified elements.

[9] Eschweiler (2012)
[10] https://silc.zih.tu-dresden.de/otf2-current/html/index.html [21.06.2016]
[11] e.g. https://silc.zih.tu-dresden.de/otf2-current/html/group__usage__writing.html [21.06.2016]