Measuring and visualizing performance

Nicolas Melot
nicolas.melot@liu.se

February 19, 2013

1 Introduction

Measuring and representing the performance of an algorithm represents quite a tedious task. Such activity requires a constant monitoring and intervention to collect data from an experiment, start the next one and check if everything works as expected. Such constant attention is error-prone. Once the data has been fully collected, it is often cumbersome and also error-prone to reshape it to a meaningful format such as a graph, or a format allowing the computation of such graph. There are many benefits in automatizing this process: making it autonomous, more reliable or faster.

The document introduces here a very simplified case study, where it is proposed to demonstrate the scalability of a multi-threaded program whose "computation" consists in achieving a given number of jumps through a loop. The sequential version takes the complete loop sequentially whereas the parallel implementation splits this loops in smaller chunks and distributes it equally among all threads. The performance of this program is monitored and plotted following a workflow and using tools introduced along the solving of this case study. The language used in this work is C, but any other programming language should work equally well. Also, the data is processed using Octave, but the scripts require little adaptation for Matlab. Section A lists and briefly describes each files provided in the script set. Section 4 introduces the problem that illustrates this case study. Section 5 details all the necessary steps to instrument a program, collect the results and use it to generate comprehensive representations. Finally, section 6 concludes the document.

2 Requirements

The user of the tools and workflow described in this document is expected to be acquainted with a number of technologies that are conjointly exploited in the flow. This includes basic knowledge of Bash, a confident use of Matlab or Octave scripting and notions in relational algebra or experience in databases[1]. Debugging or developing the tools requires extensive knowledge in Bash and/or Matlab/Octave scripts. A sharp mastery of good programming practices is always most welcome. The reader can find introductory documentation on the internet about Bash [4, 5], Make [1, 6], Matlab scripts [7] and functions [8] (also applicable to Octave[2]), the C programming language [3] and Posix thread programming [2, 9].

3 The set of scripts

All the flow described in this document makes use of a set of scripts, composed of the files in the following list:

[1]TDDD46 aims: Design and use a relational database, explain the theory behind the relational model and how this affects good design of databases.

[2]http://www.gnu.org/software/octave/

- start
  Starts the compilation process or runs all experiments in one, unsupervised batch. Start it explicitly with bash (*bash start ...*) instead of running it directly (incorrect: *./start*). This file does not need to be modified under normal use, but only for contribution purpose. Refer to the sections 5.5 and 5.6 more more information about it.

- variables
  Contains all the settings of the program and their possible values. Modify it for each different program or experiments. These values are read by *start* (see above) when compiling or running program variants. See section 5.4 to get more information about this file.

- compile
  Compile a variant of the program to be monitored. This script is called by *start* and is given as parameters all settings defined as compilation settings as well as one value for each of them, in the same order as defined in the field "compiled" of the file *variables*. Modify it to fit the program you want to monitor.

- run
  Runs a variant of the program to be monitored. It is called by *start* with compile settings first then run settings and their associated value for one program variant, given as arguments.

- settings
  Various settings read by *start* and that slightly affect its behavior. This files should not be modified nor even explicitly used.

- merge
  Merges the result of two experiments whose difference is not described in *variables*. These two experiments must have output a result matrix having the same size and the same meaning for each of its columns. This script makes use of *merge.m* and requires octave to work. Matlab cannot be used as an alternative.

- data.m, data-xxxxx-yyyyyyy.m
  Contains the raw values collected from all experiments and program variants *start* compiled and ran. This file is automatically generated when *bash start run <name>* finishes and should not be manually modified.

- plot_data.m
  Experiment-specific Octave or Matlab script that reads the result output matrix produced by the program variants and *start* when running them. This script transforms this data into smaller, ready to plot matrices. This file should be entirely re-written for all program to monitor. It can be expanded to extract other information from the raw output data.

- octave/*.m, Matlab/*.m
  Respectively Octave and Matlab specializations of data manipulation scripts, used by *plot_data.m*. Copy them into the same directory as the raw result file *data.m* and run plot_data.m from this directory (*octave plot_data.m* from the same directory, using Octave).

4 The problem

Suppose an algorithm that achieves some work in a sequential fashion, for instance the code shown in Fig.2. This sequential algorithm must be compared to an equivalent, parallel version. The parallelization of this algorithm is achieved through the division of *count* by the

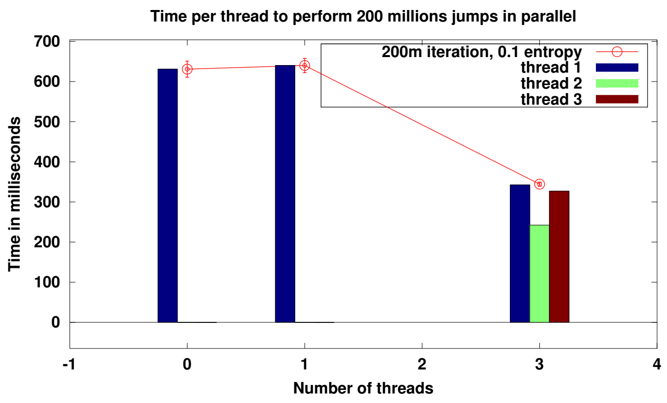

Figure 1: Example of a plot obtained through the measure and plotting scripts set.

```
static int
do_some_work(unsigned long long int count)
{
  unsigned long long int i;

  // Bring it on, yeah!
  for (i = 0; i < count; i++);

  return 0;
}
```

Figure 2: "Computation"-intensive algorithm used as case study in this tutorial.

number of threads employed, and every thread runs the loop with this new value of *count*. Since this example is very simple, the performance measurement reveal few differences from run to run. Not only this is usually not the case, but it alos prevent the illustration of the standard deviation plotting capability of this tool. Therefore *count* variates more or less, randomeley, along another property named *entropy*, which models this missing performance randomness from run to run. The parallel implementation must show good scaling properties with the number of threads employed, and demonstrate an efficient use of them. This requires the computation load to be equally spread among the threads available; in other words, all threads must compute for a roughly equivalent period of time. Both global and per-thread computation time must be recorded and shown in a representation similar to Fig.1. On this figure are represented the global computation time (red curve) as a function of number of threads, as well as individual threads' computation time for every situation, using the sequential algorithm or its parallel version using one to eight threads (the parallel version with one thread shows the overhead of parallelization). Both sequential and parallel implementations are available in appendices A.1 and A.2, respectively. These implementations introduce 3 settings: the number of threads to use (degree of parallelism), the number of jumps the program must do (that models the size of the problem) and the entropy in performance described above. Each of these settings take their value from $\mathbb{N}$, $\mathbb{R}$ and $[0;1]$, respectively. Consequently, all variants of the algorithm are contained in a 3 dimensions space. Section 5 explain the process to obtain a figure such as Fig.1.

5 Setting up an experiment, collecting, extracting and plotting data

Sections 5.1 through 5.7 describe all the necessary steps to take in order to safely monitor a program and generate a visual representation of the data gathered. All these steps take as case study the problem described in section 4.

5.1 Step 1: Set up your program

Modify your source code to integrate several versions (sequential and parallel) to the same source file. A version or another is chosen through preprocessor directives (#if, #ifdef, #ifndef, etc) at compilation time. Each directive takes a decision according to the value of definitions (defined by the directive #define or passed by the compiler). A definition represents a *setting* and takes one value among several possible. Use directives and definitions to select portions of codes that are specific to a program version, but keep these specific portions as small as possible. Figure 3 provides a simplified illustration of this technique; you can also find more example and information on the internet. Appendix A.3 shows the sequential and parallel version of the same algorithm merged into a single source file. Depending on the options given to the compiler (-D switch with gcc, see gcc –help), the sequential or a parallel version is compiled. The way the relevant options are given to the compiler is described in section 5.5.

Another way to control the behavior of a program among different versions are arguments to the program when it is started. Depending on the values in argv, the program can branch to one or another portion of code. However, since this decision is taken at runtime, it can slow down the execution time of the program. This should be avoided if the execution time of the program is monitored.

5.2 Step 2: Monitor the performance

Record the values you want to measure into variables, and display these variables at the end of the program execution, or at a moment when outputting these values does not affect the values you want to measure. Collecting data generally affects execution speed. If you want to measure execution speed, remember that printf() is very slow and will affect the performance if it is used in the middle of the code you want to measure. Consider issues such as data locality and try to make as much profit of caches, but try not to take it all from the algorithm you monitor. This might significantly slow it down.

5.3 Measure the time

There are several functions provided by C and Linux to measure execution time. In general, a good resolution is better. clock_gettime() allows the measurement of time with a precision of a nanosecond. The downside of this function is the format of this information, that takes shape in two integers, one counts seconds, the second nanoseconds. Such separate format makes difficult the fast calculation of interesting values. Later sections show that this can be done later in the process. Refer to man pages of clock_gettime() to get more information about it (type “man clock_gettime” in a Linux terminal). It can use several clocks, each of them having different properties. Below is a short description of three interesting clocks:

1. CLOCK_THREAD_CPUTIME_ID
   Computation time consumed by a thread. It does not take into account the time when the thread has been scheduled or blocked at a synchronization primitive. Consequently, it does not necessarily represent the time period between when the thread has been started and stopped. The use of this clock is safe to measure the thread execution time only when there is no synchronization involved in this thread and no

```
#if NB_THREADS > 0
  pthread_t thread[NB_THREADS];
  pthread_attr_t attr;
  thread_do_some_work_arg arg[NB_THREADS];
  int i;

  pthread_attr_init(&attr);
  for (i = 0; i < NB_THREADS; i++)
    {
      arg[i].id = i;
      // influence the number of loops to
      // achieve by the value of entropy
      // to simulate performance randomness
      arg[i].count = variate(count, entropy);

      pthread_create(&thread[i], &attr,\
 thread_do_some_work, (void*) &arg[i]);
    }

  for (i = 0; i < NB_THREADS; i++)
    {
      pthread_join(thread[i], NULL);
    }
#else
  count = variate(count, entropy);
  do_some_work(count);
#endif
```

Figure 3: If the constant NB_THREAD is defined to a value greater than zero when compiling this code, then a multi-threaded version of the code is compiled, using pthreads. Otherwise the sequential version is generated, that simply calls the function achieving the computation.

more threads are used than the underlying computer has cores unused by any other thread in the system.

2. CLOCK_PROCESS_CPUTIME_ID
Computation time consumed by the process. This is the time matching the sum of cycles consumed in parallel by all threads in the process. This time is not the time between the start and stop times of a process, equivalent to the execution time of its slowest thread. A process running two threads in parallel for 2 seconds is considered as having consumed 4 seconds of computation.

3. CLOCK_MONOTONIC
System-wise clock, that is shared between every threads. CLOCK_MONOTONIC provides a common time reference to every process or thread running in the system. This clock does not accumulate the computation time provided by parallel threads (as CLOCK_PROCESS_CPUTIME_ID does), but it does not pause when a thread is scheduled or blocked (as CLOCK_THREAD_CPUTIME_ID). It also count the time other programs running in the system consume. This is the preferred clock to measure time intervals if the computer is not already running an other computation-intensive program.

In order to calculate the time taken to execute a portion of code, $clock\_gettime()$ is called before and after each computation step being monitored. The difference between the two dates collected gives the execution time of the code between the first and the second collect of time. As described in section 5.1, these instructions should be protected by preprocessor directives, making easy to recompile and run the program with no instrumentation instructions. Figure 4 provides an example of such instrumentation and appendix B provides a fully instrumented version of the C program.

5.3.1 Choosing the right clock

In the figures 5 and 6, the same program has been compiled using the same settings, run on the same machine with the same input. Each variant has been run the same amount of time and the same script has been use to generate the plots. The only difference lies in the clock that has been used to measure the performance. The machine on which these two experiments have been run offers two cores. The difference in the measurement starts with the third core: in Fig.5, the clock used to measure the execution time is global, and measures the real time. The clock used in Fig.6 considers the time as the number of processor clock ticks that was given to threads, not real time. As the machine only offers 2 cores, a third thread has to share its core with another thread, thus making CLOCK_THREAD_CPUTIME_ID unsynchronized with real time. Notice one can use other processor or OS-provided functions to measure time. As an example, $rdtsc()$ takes profit of registers available on x86 archtectures from Pentium and returns on 64 bits the number of clock cycles elapsed since last processor reset. Figure 7 shows an implementation taking profit of the relevant register in a function a regular C program can use.

5.3.2 Output the data collected

The program whose performance is monitored must report the data in a specific way, that allows the expression of any numeric data measured. This data must take the shape of a matrix, which is represented by its values separated by a space, as shown in figure 8. In this matrix, each column represents a value of a feature measured, or information related to that value (such as the thread id corresponding to a thread's execution time). A line represents a feature's instance (one line per thread). This is the case in this example.

Figure 8 shows the output of the instrumented program of this case study. The first column represents the thread number (there are 8 threads used in this experiment), the

```
#ifdef MEASURE
  clock_gettime(CLOCK_MONOTONIC, &start);
#endif

  srandom(time(NULL));
  count = atoi(argv[1]);
  do_work(count, ENTROPY);

#ifdef MEASURE
  clock_gettime(CLOCK_MONOTONIC, &stop);

#if NB_THREADS > 0
  for (i = 0; i < NB_THREADS; i++)
    {
      printf("%i␣%i␣%li␣%i␣%li␣%i␣%li␣%i␣%li\n", i + 1,
          (int) start.tv_sec, start.tv_nsec, (int) stop.tv_sec,
          stop.tv_nsec, (int) thread_start[i].tv_sec,
          thread_start[i].tv_nsec, (int) thread_stop[i].tv_sec,
          thread_stop[i].tv_nsec);
    }
#else
  printf("%i␣%i␣%li␣%i␣%li␣%i␣%li␣%i␣%li\n", 0,
      (int)start.tv_sec, start.tv_nsec, (int)stop.tv_sec,
      stop.tv_nsec, (int)thread_start.tv_sec,
      thread_start.tv_nsec, (int)thread_stop.tv_sec,
      thread_stop.tv_nsec);
#endif
#endif

  // Always report a successful experiment
  return 0;
}
```

Figure 4: *clock_gettime*() before and after the portion of code being monitored collects the necessary data to compute the time used to run this portion of a program.

```
static void*
thread_do_some_work(void* arg)
{
  thread_do_some_work_arg *args;
  args = (thread_do_some_work_arg*) arg;

#ifdef MEASURE
  clock_gettime(CLOCK_MONOTONIC,\
 &thread_start[args->id]);
#endif

  do_some_work(args->count);

#ifdef MEASURE
  clock_gettime(CLOCK_MONOTONIC,\
 &thread_stop[args->id]);
#endif

  return NULL;
}
```

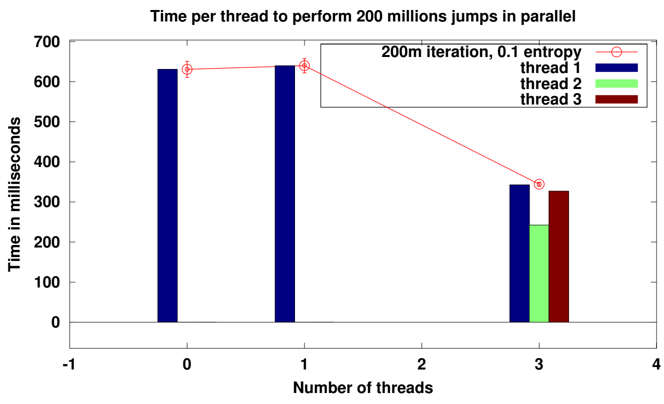

Figure 5: Code using CLOCK_MONOTONIC to monitor time, and performance measured from the data collected.

```
static void*
thread_do_some_work(void* arg)
{
  thread_do_some_work_arg *args;
  args = (thread_do_some_work_arg*) arg;

#ifdef MEASURE
  clock_gettime(CLOCK_THREAD_CPUTIME_ID,\
 &thread_start[args->id]);
#endif

  do_some_work(args->count);

#ifdef MEASURE
  clock_gettime(CLOCK_THREAD_CPUTIME_ID,\
 &thread_stop[args->id]);
#endif

  return NULL;
}
```

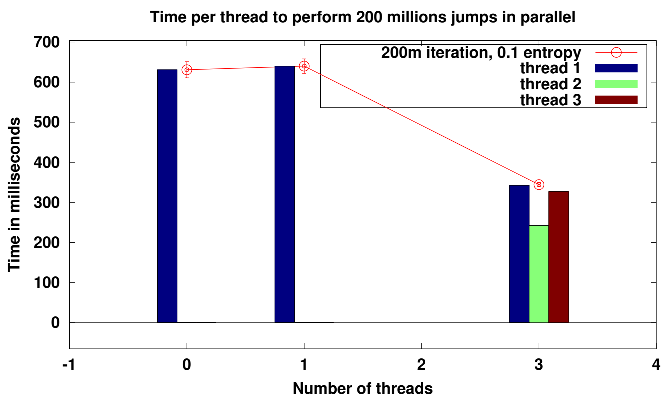

Figure 6: Code using CLOCK_THREAD_CPUTIME_ID to monitor time, and performance measured from the data collected.

```
#ifdef __i386
extern __inline__ uint64_t rdtsc() {
  uint64_t x;
  __asm__ volatile ("rdtsc" : "=A" (x));
  return x;
}
#elif defined __amd64
extern __inline__ uint64_t rdtsc() {
  uint64_t a, d;
  __asm__ volatile ("rdtsc" : "=a" (a), "=d" (d));
  return (d<<32) | a;
}
#endif
```

Figure 7: C inline assembly implementation of rdtsc using hardware registers.

```
1 28689 94... 28690 27... 28689 942207530 28690 270489437
2 28689 94... 28690 27... 28689 942220041 28690 250966917
3 28689 94... 28690 27... 28689 948509233 28690 254110653
4 28689 94... 28690 27... 28689 948318059 28690 269370024
5 28689 94... 28690 27... 28689 948004682 28690 270454439
6 28689 94... 28690 27... 28689 943606755 28690 264533334
7 28689 94... 28690 27... 28689 942284021 28690 265217907
8 28689 94... 28690 27... 28689 956254814 28690 268802904
```

Figure 8: Format the data must take when output from an instrumented program.

second column is the time in seconds when the algorithm was started and the third is the time in nanoseconds (added to the seconds of the second column) when the program was started. The next two columns represent in the same format the time when the program stopped. The four next values represent the start and stop time for every thread.

Notice that the four columns from 2 to 5 are identical in every line, as this information is global to the program and not thread-specific. Columns 6 to 9 are different in each line, since each of them show the execution time of one thread. The reason why global data is repeated is because the format of a matrix *must* be preserved, in order to be parsed in further steps. In addition, all values must be numeric, integer or decimal.

Once data is produced or in case of any failure, it is important to exit the program with a relevant exit value (*return 0*). Returning 0 reports a run to work as expected and output to be taken into account. A return value different than 0 denotes a unsuccessful run and will make the batch to discard output produced. See section 5.6 for more details about return values and the way they are handled.

5.4 Step 3: Set up the different compilation and runtime settings to be used

The scripts use the content of the file *variables* to build the complete list of possible versions the program can have, and that must be compiled. This list of versions is also used to run and collect data from all possible versions. The settings' names and the values they can take are written using the syntax for Bash scripts. As discussed in section 5.1, settings' values can be given as programs' runtime arguments or preprocessor directives resolved at compilation time. This distinction must be made when filling settings' names and possible values in *variables*. Appendix C gives a complete example of this file. Several variables need to be filled:

- output
  Describes the meaning of each column of the matrix the program outputs when sending the data it measured (one word per column). Elements are separated by a space and the list must be surrounded by double quotes(").

- compile
  List of the settings whose value is given at compilation time to the C preprocessor. All elements are separated by a space and the list is surrounded by *parentheses* only (no double quotes). Elements of this list cannot take some name such as *count*, as these names are used in other parts of the scripts and their use can randomly alter the good functioning of the scripts. There is no exhaustive list of these forbidden values, so users are encouraged to report them.

- run
  List of the settings whose value is given at runtime to the program as an argument.

```
CFLAGS=-g -O0 -Wall -lrt -pthread -DMEASURE\t
 -DNB_THREADS=$(NB_THREADS) -DENTROPY=$(ENTROPY)
[...]
        gcc $(CFLAGS) -o program$(SUFFIX) program.c
```

Figure 9: GCC is invoked with options defining symbols handled by the C preprocessor directives (#if, etc).

> This field has the same limitations and constraints as *compile*. The names used in *compile* and *run* are mutually exclusive.

Every element in *run* and *compile* lists must be defined as a variable, taking as values the list of all possible value this setting can take. This list must be surrounded by double quotes.

5.5 Step 4: Compile your program

It is a good practice to provide a Makefile to build your program. The syntax of a Makefile is not described in this document. Refer to make's manual[3] for this, or browse the internet for examples. Section D.2 gives the one used in this case study and Fig.9 shows the important lines in the context of this section. What is important to know here is that make and Makefiles can take arguments of the form *argument* = *value* when invoking the make tool. This argument is interpreted as a variable taking this value in the Makefile. Figure 9 shows that gcc is called in order to compile program.c to a binary output whose name depends on a suffix given when make is invoked. GCC also takes as arguments CFLAGS, that defines the constant symbols MEASURE, NB_THREADS and ENTROPY. The two latter symbols take respectively the value of the makefile's variables NB_THREADS and ENTROPY, which are given as arguments when invoking make. CFLAGS also includes switches such as -Wall that forces a more strict control on the code and -O0 which disables every optimizations the compiler could provide. The former flag is a good programming practice while the latter is important to make sure that no unexpected optimization can alter the behavior of the code, thus allowing more accurate performance measurements. -g and other flags enable debugging and provide with the linker the libraries that the program has to be linked with.

More focus in this case study is given to the file *compile*, as this file is called once per variant to be compiled with the list of compilation settings and one possible combination given as arguments. A complete version of this file written in Bash is given in appendixD and its most important parts are illustrated in Fig.10. The arguments are given in the same order as the list described in section 5.4. In this case study, $1 defines the entropy and in $2 is given the number of threads that have to be compiled in this version. It is a good practice to assign these values to variables whose name is comprehensive. This clarifies what information is used in every line of the script, and facilitates the propagation of changes from *variables* to *compile*. Figure 10 shows that make is called with arguments defining ENTROPY to the value of the entropy setting for this compilation as well as the number of threads to use. At the binary output's file name is appended the value of entropy and number of threads. Having a separate name for each compiled program variants prevents variant from being overwritten by each other, and makes it possible to start the right version when a particular measurement is to be started. Notice *compile* must return a value so the global batch knows if it succeed or not. This is achieved with *exit <value>* where *<value>* is the value to return. 0 denote success and anything else report a failure. A convenient way to obtain a suitable value consists in reading *$?*, which returns same value as the last program or command called, usually having the same convention.

[3] http://www.gnu.org/s/make/

```
entropy=$1
nb_threads=$2

make ENTROPY=$entropy NB_THREADS=$nb_threads \
  SUFFIX=-$entropy-$nb_threads
```

Figure 10: *compile* catches the values of compilations settings and passes them to make.

```
#!/bin/bash -f

# compilation settings
entropy=$1
nb_threads=$2

# run settings
ct=$3
try=$4

./program-$entropy-$nb_threads $ct

exit ?
```

Figure 11: *run* catches the compilation settings first, then the run settings and run the right binary with relevant parameters.

Once the files *variables* and *compile* are ready, the compilation process is started with the command *bash start compile*.

5.6 Step 5: Run your experiments

All algorithm's variants are run through the bash script *run*, that, similarly to *compile* (see section 5.5) takes as arguments one possible combination of all the settings each time it is called. An example of this file is given in appendix E and illustrated in Fig. 11. The compilation settings are given first then the runtime settings are appended, both taken from *variables*. In this example, arguments 1 through 2 are taken from the compilation settings and arguments 3 and 4 represent the runtime settings. The binary file whose file name has for suffix the matching entropy value and number of threads is executed with the runtime argument ct, that defines the number of jumps the program has to perform. The parameter *try* is never used and stands only to run several times (defined in *variables* as 1 to 10) the same variant.

In order to make the scripts catch their output from the measurements, *run* must absolutely print on standard output the matrix generated by the program variant, regardless of the way it outputs the data collected. *run* must output this matrix, and nothing else, to standard output. On this example, the program variant directly prints it on the terminal. It may also display data files the program would generate, after some possible filtering. As for *compile* script (see section **??**), it is very important the *run* script returns 0 if the experiment was successful or another value in case of failure. The use the keyword *exist* and special variable *$?* is suitable to this purpose.

Once the file *variables* is ready and the compilation process is finished, all program variants are run and by the command *bash start run* $<$ *name* $>$, where $<$ *name* $>$ is a name to the experiment being conducted. The script catches the data outputted on standard output, and stores it into the file *data-$<$ name $>$-$<$ date $>$.m*. The batch script tries to give a very rough estimation of the remaining time before the batch is finished. This evaluation is based on the time that was necessary to run the program versions already run and is calculated using integer seconds only. It is therefore very rough and shouldn't be used to estimate the perfomance of the program monitored. If a variant run did not work properly and *run* reported this failure through exit value, then the batch will record its output and neutralize it through comments and warnings in the final data file. As a consequence, failures do not necessarily jeopardize the whole measurement process, since their output are identified, neutralized. The faulty variant can be fixed and run in a later batch using the same experiment name. Alternatively, since incorrect output is not entirely discarded but only commented, the user can still collect it manually and analyze it if necessary.

If two experiments are successively run with the same name, their results are merged together into a single file named *data.m*. Every experiment runs separately stores its result in a specific output file. The results of two experiments of the same name can be merged again to *data.m* using the command *bash start merge* $<$ *name* $>$. Two different experiments, that differ in features that are not defined by any setting in *variables* can be merged together by the command *bash merge* $<$ *res*1*.m* $>$ $<$ *res*2*.m* $>$, to a file name *MERGE-* followed by the two file names. In this file, data coming from one or another experiment can be identified thanks to an additional column inserted before the first column or original data. It takes as value the index of the arguments given to *merge* that referred to its original result file, starting at 0. For instance, in the output of *merge exp1.m exp2.m*, all data found in exp1.m will be prefixed by 0 and all data coming from exp2.m is prefixed by 1 in the output merged file. Note that this feature requires Octave installed, and that Matlab does not support such operation.

5.7 Step 6: Process and plot the data collected

The data generated when running all program variants is a numeric matrix that Matlab or Octave can directly load and manipulate as well as columnns' name, both wrapped in a Cell. In order to ease the manipulation, several functions inspired from relational algebra are provided. A complete commented example for this case study is available in the file *plot_data.m*. This script run by Matlab or Octave collects the data and processes it to obtain specific results used to plot graphs. It relies mainly on a set of data manipulation routines shortly described below. This script is usually very dense regarding the meaning of its lines, which makes it difficult to write correctly or to find the source of its defects. It usually takes shape of three parts: the first part collects the data, filters it through *select* and *where* operations and transforms it by applying custom functions. In this example such functions take recorded start and stop times (global and per thread) through their values in seconds and nanoseconds and compute the time in millisecond between these two steps in the program run. The second part extracts and reshapes this data in order to obtain small matrices ready to be plot. Finally, the third part uses the plot functions to generate and format graphs, and stores them in graphic files.

Below is a very short description of the functions available. More extensive documentation is available in the files where these functions are implemented:

- select
  Filters the input matrix and keeps only the columns whose indexes are given as parameters (select.m).

- duplicate
  Duplicates once or more one or several columns of the input matrix.

- where
  Select matrices' row that fulfills conjunctive and/or disjunctive conditions (where.m)

- apply
  Applies a custom function and writes its result to the columns which index is given in arguments (apply.m).
- groupby
  Separates and groups rows by the content of the columns whose index is given as argument (groupby.m)
- reduce
  Reduces groups produced by groupby to one row per group, using reduction functions such as mean, standard deviation or custom functions. Merges rows to a matrix (reduce.m)
- extend
  Makes sure every groups has the same number of rows and inserts new rows with suitable data if necessary (extend.m)

Matrices can be plotted using the following plotting routines:

- quickplot
  Plots one or more curves into a graph and displays axis labels, curves legend and graph title in a flexible way (quickplot.m)
- quickerrorbar
  Same as quickplot, but also allows the plotting of error bars (quickerrorbar.m)
- quickbar
  Plots a histograms with one or more bars per element in the x axis (quickbar.m)
- quickgantt
  Draws a gantt diagram, one horizontal line per element (quickgantt.m)

The command *octave plot_data.m* runs the octave script named *plot_data.m* and generates the graphs. Matlab cannot run the script in the same way. Open Matlab and browse in the left frame until you reach the folder where the results and all the scripts are stored, then start the script "plot_data.m". The example used as a case study in this document produces the graph shown in Fig.1. Its shows the time to perform 200 millions jumps using a sequential loop or splitting it between 1 to 8 threads. The plots clearly show that the global time decreases with the number of threads and it shows that the computation load is distributed equally among the threads. Finally, the low difference between the sequential and the parallel version using one thread shows that the overhead of parallelization is negligible compared to the overall performance.

6 Conclusion

This documents describes a general methodology to manage easily the run of several variants of a single algorithm implementation, and the collection, processing and representation of collected data. This is a tedious task that the tools described here make both easier and faster. Although they considerably reduce the occasions to commit mistakes in experiments due to human interventions, they do not eliminate such mistake. One possible remaining pitfall lies in the data collection method (the choice of the clock) and the parts that could not be automated. A too big confidence in this automated process may hinder cautions to take when performing the remaining manual interventions, or even simply makes the user to forget these steps even if they may be important. More efforts need to be done to improve the set of scripts described here, such as providing them with the ability to interrupt and resume an experiment. Contributions are encouraged, both through coding more functionality or bug reports.

A Program to monitor performance

A.1 Sequential version

```
#include <stdio.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

/* to be defined at compile-time:
 * ENTROPY
 */

static int
do_some_work(unsigned long long int count)
{
  unsigned long long int i;

  // Bring it on, yeah!
  for (i = 0; i < count; i++)
    ;

  return 0;
}

static unsigned long long int
variate(unsigned long long int count, float entropy)
{
  unsigned long long int variation, res;
  variation = (unsigned long long int) ((((float) (random() % RAND_MAX))
 / RAND_MAX) * count
        * entropy);

  res = (unsigned long long int) (((float) (count + variation))
 / (NB_THREADS
        == 0 ? 1 : NB_THREADS));

  return res;
}

static int
do_work(unsigned long long int count, float entropy)
{
  count = variate(count, entropy);
  do_some_work(count);

  return 0;
}

int
main(int argc, char ** argv)
{
  unsigned long long int count;
  srandom(time(NULL));
  count = atoi(argv[1]);
  do_work(count, ENTROPY);

  return 0;
}
```

A.2 Multithreaded version

```
#include <stdio.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

/* to be defined at compile-time:
 * NB_THREADS
 * ENTROPY
 */

typedef struct
{
  int id;
  unsigned long long int count;
} thread_do_some_work_arg;

static int
do_some_work(unsigned long long int count)
{
  unsigned long long int i;

  // Bring it on, yeah!
  for (i = 0; i < count; i++)
    ;

  return 0;
}

static void*
thread_do_some_work(void* arg)
{
  thread_do_some_work_arg *args;
  args = (thread_do_some_work_arg*) arg;

  do_some_work(args->count);

  return NULL;
}

static unsigned long long int
variate(unsigned long long int count, float entropy)
{
  unsigned long long int variation, res;
  variation = (unsigned long long int) ((((float) (random()
 % RAND_MAX)) / RAND_MAX) * count
      * entropy);

  res = (unsigned long long int) (((float) (count +
 variation)) / (NB_THREADS
      == 0 ? 1 : NB_THREADS));

  return res;
}

static int
do_work(unsigned long long int count, float entropy)
{
  pthread_t thread[NB_THREADS];
  pthread_attr_t attr;
  thread_do_some_work_arg arg[NB_THREADS];
  int i;

  pthread_attr_init(&attr);
  for (i = 0; i < NB_THREADS; i++)
    {
      arg[i].id = i;
      arg[i].count = variate(count, entropy);

      pthread_create(&thread[i], &attr,
thread_do_some_work, (void*) &arg[i]);
    }

  for (i = 0; i < NB_THREADS; i++)
    {
      pthread_join(thread[i], NULL);
    }

  return 0;
}

int
main(int argc, char ** argv)
{
  unsigned long long int count;

  srandom(time(NULL));
  count = atoi(argv[1]);
  do_work(count, ENTROPY);

  return 0;
}
```

A.3 Unified sequential and multithreaded versions

```
#include <stdio.h>
#include <time.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <pthread.h>

/* to be defined at compile-time:
 * NB_THREADS
 * ENTROPY
 */

#if NB_THREADS > 0
typedef struct
{
  int id;
  unsigned long long int count;
} thread_do_some_work_arg;
#endif

static int
do_some_work(unsigned long long int count)
{
  unsigned long long int i;

  // Bring it on, yeah!
  for (i = 0; i < count; i++)
    ;

  return 0;
}

#if NB_THREADS > 0
static void*
thread_do_some_work(void* arg)
{
  thread_do_some_work_arg *args;
  args = (thread_do_some_work_arg*) arg;

  do_some_work(args->count);

  return NULL;
}
#endif

static unsigned long long int
variate(unsigned long long int count, float entropy)
{
  unsigned long long int variation, res;
  variation = (unsigned long long int) ((((float) (random()
 % RAND_MAX)) / RAND_MAX) * count
      * entropy);

  res = (unsigned long long int) (((float) (count +
 variation)) / (NB_THREADS
      == 0 ? 1 : NB_THREADS));

  return res;
}

static int
do_work(unsigned long long int count, float entropy)
{
#if NB_THREADS > 0
  pthread_t thread[NB_THREADS];
  pthread_attr_t attr;
  thread_do_some_work_arg arg[NB_THREADS];
  int i;

  pthread_attr_init(&attr);
  for (i = 0; i < NB_THREADS; i++)
    {
      arg[i].id = i;
      arg[i].count = variate(count, entropy);

      pthread_create(&thread[i], &attr,
 thread_do_some_work, (void*) &arg[i]);
    }

  for (i = 0; i < NB_THREADS; i++)
    {
      pthread_join(thread[i], NULL);
    }
#else
  count = variate(count, entropy);
  do_some_work(count);

#endif
  return 0;
}

int
main(int argc, char ** argv)
{
  unsigned long long int count;

  srandom(time(NULL));
  count = atoi(argv[1]);
  do_work(count, ENTROPY);

  return 0;
}
```

B Fully instrumented C program

```
#include <stdio.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
```

```
/* to be defined at compile-time:
 * NB_THREADS
 * ENTROPY
 */

#if NB_THREADS > 0
typedef struct
{
  int id;
  unsigned long long int count;
} thread_do_some_work_arg;
#endif

#ifdef MEASURE
#if NB_THREADS > 0
struct timespec thread_start[NB_THREADS],
 thread_stop[NB_THREADS];
#else
struct timespec thread_start, thread_stop;
#endif
#endif

static int
do_some_work(unsigned long long int count)
{
  unsigned long long int i;

  // Bring it on, yeah!
  for (i = 0; i < count; i++)
    ;

  return 0;
}

#if NB_THREADS > 0
static void*
thread_do_some_work(void* arg)
{
  thread_do_some_work_arg *args;
  args = (thread_do_some_work_arg*) arg;

#ifdef MEASURE
  clock_gettime(CLOCK_MONOTONIC, &thread_start[args->id]);
#endif

  do_some_work(args->count);

#ifdef MEASURE
  clock_gettime(CLOCK_MONOTONIC, &thread_stop[args->id]);
#endif

  return NULL;
}
#endif
```

```
static unsigned long long int
variate(unsigned long long int count, float entropy)
{
  unsigned long long int variation, res;
  variation = (unsigned long long int) ((((float) (random()
 % RAND_MAX)) / RAND_MAX) * count
      * entropy);

  res = (unsigned long long int) (((float) (count +
 variation)) / (NB_THREADS
      == 0 ? 1 : NB_THREADS));

  return res;
}

static int
do_work(unsigned long long int count, float entropy)
{
#if NB_THREADS > 0
  pthread_t thread[NB_THREADS];
  pthread_attr_t attr;
  thread_do_some_work_arg arg[NB_THREADS];
  int i;

  pthread_attr_init(&attr);
  for (i = 0; i < NB_THREADS; i++)
    {
      arg[i].id = i;
      arg[i].count = variate(count, entropy);

      pthread_create(&thread[i], &attr, thread_do_some_work,
 (void*) &arg[i]);
    }

  for (i = 0; i < NB_THREADS; i++)
    {
      pthread_join(thread[i], NULL);
    }
#else

#ifdef MEASURE
  clock_gettime(CLOCK_MONOTONIC, &thread_start);
#endif

  count = variate(count, entropy);
  do_some_work(count);

#ifdef MEASURE
  clock_gettime(CLOCK_MONOTONIC, &thread_stop);
#endif

#endif
  return 0;
}

int
main(int argc, char ** argv)
{
  unsigned long long int count;
#ifdef MEASURE
#if NB_THREADS > 0
  int i;
#endif
  struct timespec start, stop;
#endif

#ifdef MEASURE
  clock_gettime(CLOCK_MONOTONIC, &start);
#endif

  srandom(time(NULL));
  count = atoi(argv[1]);
  do_work(count, ENTROPY);

#ifdef MEASURE
  clock_gettime(CLOCK_MONOTONIC, &stop);

#if NB_THREADS > 0
  for (i = 0; i < NB_THREADS; i++)
    {
      printf("%i %i %li %i %li %i %li %i %li\n", i + 1,
          (int) start.tv_sec, start.tv_nsec, (int) stop.tv_sec,
          stop.tv_nsec, (int) thread_start[i].tv_sec,
          thread_start[i].tv_nsec, (int) thread_stop[i].tv_sec,
          thread_stop[i].tv_nsec);
    }
#else
  printf("%i %i %li %i %li %i %li %i %li\n", 0,
      (int)start.tv_sec, start.tv_nsec, (int)stop.tv_sec,
      stop.tv_nsec, (int)thread_start.tv_sec,
      thread_start.tv_nsec, (int)thread_stop.tv_sec,
      thread_stop.tv_nsec);
#endif
#endif

  // Always report a successful experiment
  return 0;
}
```

C Variables

```
#!/bin/bash -f

output="thread start_time_sec start_time_nsec stop_time_sec\
stop_time_nsec thread_start_sec thread_start_nsec thread_stop_sec\
 thread_stop_nsec"
```

```
# Never use these names for run or compile features
# count

run=(ct try)

ct="100000000␣200000000"              # amount of data to be\
 written to main memory
try=`seq 1 10`                        # Number of different run\
 per setting

compile=(entropy nb_threads)
entropy="0.1"                         # Randomness in the\
 calculation time
nb_threads=`seq 0 8`
```

D Compilation

D.1 Compile script

```
#!/bin/bash -f

# Compile settings
entropy=$1
nb_threads=$2

make ENTROPY=$entropy NB_THREADS=$nb_threads\
 SUFFIX=-$entropy-$nb_threads

exit $?
```

D.2 Makefile

```
NB_THREADS=1
ENTROPY=0.01
SUFFIX=

CFLAGS=-g -O0 -Wall -lrt -pthread -DMEASURE\
 -DNB_THREADS=$(NB_THREADS) -DENTROPY=$(ENTROPY)

TARGET=program$(SUFFIX)

all: $(TARGET)

clean:
	$(RM) program
	$(RM) program-*
	$(RM) *.o

$(TARGET): program.c
	gcc $(CFLAGS) -o program$(SUFFIX) program.c
```

E Run script

```
#!/bin/bash -f

# compilation settings
entropy=$1
nb_threads=$2

# run settings
ct=$3
try=$4

./program-$entropy-$nb_threads $ct

exit 0
```

F Data processing and representation script

This file is to big to fit in this document. Please open the file directly.

F.1 Compute global time

```
function y = time_difference_global(col, row)
        NSEC_IN_SEC = 1000000000;
        MSEC_IN_SEC = 1000;
        NSEC_IN_MSEC = NSEC_IN_SEC / MSEC_IN_SEC;
        y = (((row(7) - row(5)) * NSEC_IN_SEC + row(8)
 - row(6))) / NSEC_IN_MSEC;
endfunction
```

F.2 Compute per-thread time

```
function y = time_difference_thread(col, row)
        NSEC_IN_SEC = 1000000000;
        MSEC_IN_SEC = 1000;
        NSEC_IN_MSEC = NSEC_IN_SEC / MSEC_IN_SEC;
        y = (((row(12) - row(10)) * NSEC_IN_SEC + row(13)
 - row(11))) / NSEC_IN_MSEC;
endfunction
```

References

[1] Anonymous. An introduction to the unix make utility. http://frank.mtsu.edu/ csdept/-FacilitiesAndResources/make.htm. Last accessed: 2012-08-06.

[2] B. Barney. Posix threads programming. https://computing.llnl.gov/tutorials/pthreads/, 2012. Last accessed: 2012-08-06.

[3] Cprogramming.com. C tutorial. http://www.cprogramming.com/tutorial/c-tutorial.html. Last accessed: 2012-08-06.

[4] Mike G. Bash programming - introduction how-to. http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html, 2000. Last accessed: 2012-08-06.

[5] M. Garrels. Bash guide for beginners. http://tldp.org/LDP/Bash-Beginners-Guide/html/, 2008. Last accessed: 2012-08-06.

[6] GNU. Gnu 'make'. http://www.gnu.org/software/make/manual/make.html. Last accessed: 2012-08-06.

[7] T. Huber. Introduction to matlab scripts. http://physics.gac.edu/ huber/envision/tutor2/mtlscrip.htm, 1997. Last accessed: 2012-08-06.

[8] G. Recktenw. Matlab functions – basic features. http://web.cecs.pdx.edu/ gerry/MATLAB/programming/basics.html, 1995. Last accessed: 2012-08-06.

[9] Tim. pthreads in c – a minimal working example. http://timmurphy.org/2010/05/04/pthreads-in-c-a-minimal-working-example/, 2010. Last accessed: 2012-08-06.