The EuroBen Throughput Benchmark Framework

A.J. van der Steen

Computational Physics
Utrecht University
The Netherlands

Abstract

In this report we present a framework that can be used to develop a large-scale throughput benchmark. It includes programs and shell scripts as well as a predefined directory structure to embed test programs in. In addition, an interactive benchmark that is included in this framework is described. The EuroBen synthetic benchmark set is included as a predefined program [2] sequence for easy reference for various machines.

Contents

1 Introduction

To acquire an update or replacement of a large computer system that is used by an extensive and varied user community benchmarking the individual processor and network performance is necessary but not sufficient. To gain insight into the multi-user capacity a throughput benchmark that reflects the needs of the present — and projected — workload is mandatory.
Because such a workload will differ from site to site it is not sensible to develop one static benchmark that should be run by every institution where a major replacement is due[1]. However, a *framework* can be provided in which the appropriate programs can be inserted and that greatly eases the burden of developing such a benchmark from scratch.
The EuroBen Throughput Benchmark Framework presented here tries to make putting together a complete throughput benchmark for general Unix-based less cumbersome. It assumes that in the complete benchmark the following components are present and will be run:

- A Single-User benchmark that enables running all the constituent programs in the benchmark individually and records the results in distinct run logfiles.
- A throughput multi-user benchmark of which the constituent programs are scheduled through the control of a utility that injects them into the batch queueing system at times and with multiplicities defined by the benchmarker.
- An interactive benchmark that can be run in single and multiple sessions per workstation and from multiple workstations. This benchmark can (and should) be run on an empty system and while running the throughput component.
  Response times of the interactive commands are separately logged and some simple utilities are present to gather statistics on them.

This report decribes these three parts in detail in addition to specific information of the programs. The installation and running of the Single-User benchmark and the Throughput benchmark are described together as they have much of the procedures in common. The Interactive benchmark is more or less independent and the installation and operation are somewhat different from the those of the other parts. Therefore the Interactive benchmark is described in a separate part. A part of the interactive benchmark consists of running it concurrently with the Throughput benchmark. However, this has no influence on either of the parts.

As already remarked, for the purpose of performing the Single-User benchmark and Throughput benchmark, a software package was developed which to a large extent automates these benchmark tests. The framework consists of a number of benchmark (utility) files and a series of test modules. The foremost utility is the `make` utility, which enables one to build and run modules by giving a few simple commands. The original user programs can be integrated in the benchmark framework in a more or less uniform way. The user programs should be adapted for the benchmark: one should remove as much as possible machine dependencies from the original user programs. For the programs that for some reason the generation of the executables and/or checking of the output cannot be done fully automatic this is should be pointed out in the individual `README` files that should accompany each program.

The following three sections of this document describe the structure and operation of the Single-User benchmark and the Throughput benchmark. Section 6 deals with the information about the programs in the benchmark. Because this package is in principle only a framework, this description only pertains to the EuroBen Benchmark (presently Version 3.9) [2] that is, apart from the throughput benchmark, the only constant constituent in the framework. It consists of about 25 programs in Fortran 77/90 ranging from testing basic operations to application kernels that provide a good insight in the capabilities of individual processors in single-user mode. Section 7 describes the interactive benchmark. The output of a sample run of the interactive benchmark is attached as an appendix. Lastly, section 8 describes the run rules and which runs may be performed by the benchmarkers and/or vendors. The package is distributed as a g-zipped `tar` file that can be downloaded from:

`www.phys.uu.nl/~steen/euroben/programs/`, file `euroframe.tar.gz`.

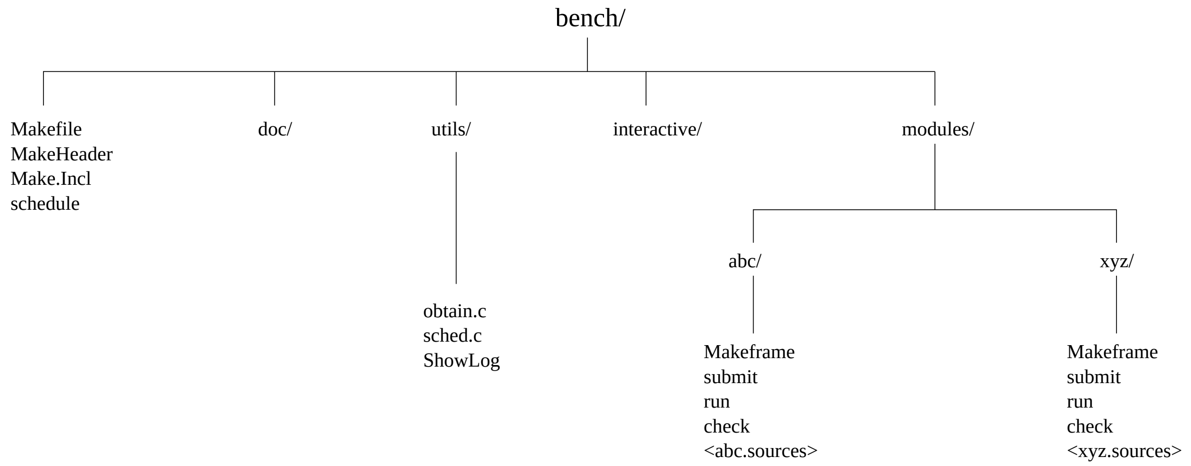

Figure 1: *Structure of the benchmark. Directory names are followed by a slash.*

2 Preliminaries for setting up the benchmark

The "filling" of the framework with programs that constitute a meaningful benchmark is still a large task. The first and foremost action is to select the appropriate programs, that (will) reflect the workload on the new system. Once the programs have been selected, the following steps should be taken:

1. Test each program at least on two different systems to minimise the occurence of machine-dependencies.
2. Use the invariant part of the makefiles as the `Makeframe`s for each of the programs in the benchmark (see section 3).

3 Structure of the benchmark

The benchmark's general structure is as shown in Figure 1 (assuming that there are only two modules, `abc` and `xyz`): The root directory need not be called `bench/`. In the examples given in this report, `${BENCHMARKROOT}` will refer to the benchmark's root directory.

3.1 The root directory

The top-level `Makefile` contains macros and targets for building the modules, for running them, and for collecting the relevant results. The most relevant targets in `Makefile` are:

`make`:	for each module, builds the required Makefile.
`build_all`:	for each module, compiles the source code; also compiles `obtain.c` and `sched.c`.
`run_all`:	runs all modules, in batch mode.
`collect_all`:	for each module, collects all results in a separate directory.
`clean`:	for each module, removes object code.
`clobber`:	for each module, removes object code, first-step results, executable, and `Makefile`.
`rmres1`:	for each module, removes first-step results and temporary directories.
`rmres2`:	for each module, removes second-step results (those resulting from making the `collect_all` target).
`sched`:	runs a number of jobs, as specified in a configuration file, `schedule`.

In order to be able to make these targets, the current directory should always be the benchmark's root directory.

MakeHeader is a shell script which prepends, for each module, generic file Make.Incl to module-specific file Makeframe in a file Makefile. Make.Incl is a listing of machine-dependent environment variables like compiler names and compilation options, libraries, etc.

3.2 The utilities

File obtain.c is used for obtaining, interactively, descriptive information on the current benchmark run. File sched.c is used in the context of a throughput benchmark: it schedules jobs according to the specifications in configuration file schedule.

These utilities can be built by hand by entering the following commands:

```
cd ${BENCHMARKROOT}/utils
make obtain
make sched
```

Another utility is ShowLog. The use of this utility is explained in section 3.8. ShowLog is a shell script and consequently does not have to be built.

3.3 The modules

Each module resides in a separate directory bearing the module's name (often, the module's main executable has the same name as well). Thus, module abc (which has an executable called abc) resides in directory bench/modules/abc. Apart from its source files, a module's directory contains four additional files: a preliminary makefile, Makeframe, and three shell scripts, submit, run, and check. The first file is discussed in the section "How to build a module", 3.4; submit and run are discussed in the section "Setting up the environment", 3.9. File check is called on the fly when the execution of a module is finished. It extracts those parts from the output files that contain data relevant for the purpose of checking for correctness and shows it in the jobs standard output. In case of a single-user benchmark the run script saves all output files of the job, in case of a throughput benchmark only the jobs standard output and standard error are saved.

3.4 How to build a module

For each module, a module-specific Makefile is required. This file is constructed by making target make, which prepends the machine-dependent makefile part, file Make.Incl, to the machine-independent part, file Makeframe. The command to be given is simply:

```
make make
```

After the required Makefiles have been built, each module's source code has to be compiled, and executables have to be generated. There are three ways of doing this. Of course the Makeframes have to be provided by the benchmarker for each code inserted into the framework.

3.4.1 Building a single module from the module's Makefile

For building a single module, the root Makefile is not required. For building, say, module abc, one can change to directory modules/abc:

```
cd ${BENCHMARKROOT}/modules/abc
```

and enter the command:

```
make install
```

Notice that utilities obtain and sched are not built: only target build_all will do this. Of course, also these utilities can be built by hand, as shown in section 3.2.

3.4.2 Building a single module root Makefile

For building a single module, say abc, the following command suffices:

```
make abc_bld
```

Notice that utilities obtain and sched are not built: only target build_all will do this. Of course, these utilities can be built by hand, as shown in section 3.2. Further notice the suffix "_bld": this is to distinguish the target from run and collect targets.

3.4.3 Building all modules from the root Makefile

To build all the modules in one go, the following command suffices:

```
make build_all
```

The files obtain.c and sched.c will be compiled on the fly, yielding executables obtain and sched.

3.5 How to run a module

There are a number of different objectives to be distinguished. These are dealt with in the next three subsections.

3.5.1 Running a single module from run script

One can directly run a module (i.e., not as a batch job), say abc, by entering the following commands:

```
BENCHTYPE=single
BATCHEXT=.0001
BENCHOUTPUT=BENCH_OUTPUT
LOGFILE=${BENCHMARKROOT}/logfile
STARTDIR=${BENCHMARKROOT}/modules/abc
export BATCHEXT BENCHOUTPUT LOGFILE STARTDIR
${BENCHMARKROOT}/modules/abc/run
```

Environment variable BENCHTYPE specifies the kind of benchmark being run. Variable BATCHEXT is used for uniquely identifying a certain run of a certain (group of) module(s). Variables BENCHOUTPUT, LOGFILE, and STARTDIR should always have the values given in the example. BENCH_OUTPUT is the directory in which a module's results will initially be stored (the so-called first-step results). The file logfile contains information on the run. STARTDIR is the module's main directory. Shell script run will check the environment variables and run the module. After having issued the above command, the directory structure has become as shown in Figure 2.

3.5.2 Running a single module from the root Makefile

For running a single module, say abc, in an NQS batch job, one issues the command

```
make abc_run BENCHTYPE=<benchtype> BATCHEXT=<batchext>
```

The environment variable BENCHTYPE represents the type of benchmark currently being run (presumably either "single" or "thruput"), and is used for determining the parameters required for submitting the job; these parameters are selected in each module's submit script. BATCHEXT should, preferably, consist of a dot followed by four digits; it is appended to the name of each output file of the current run, thus generating uniquely identifiable file names.

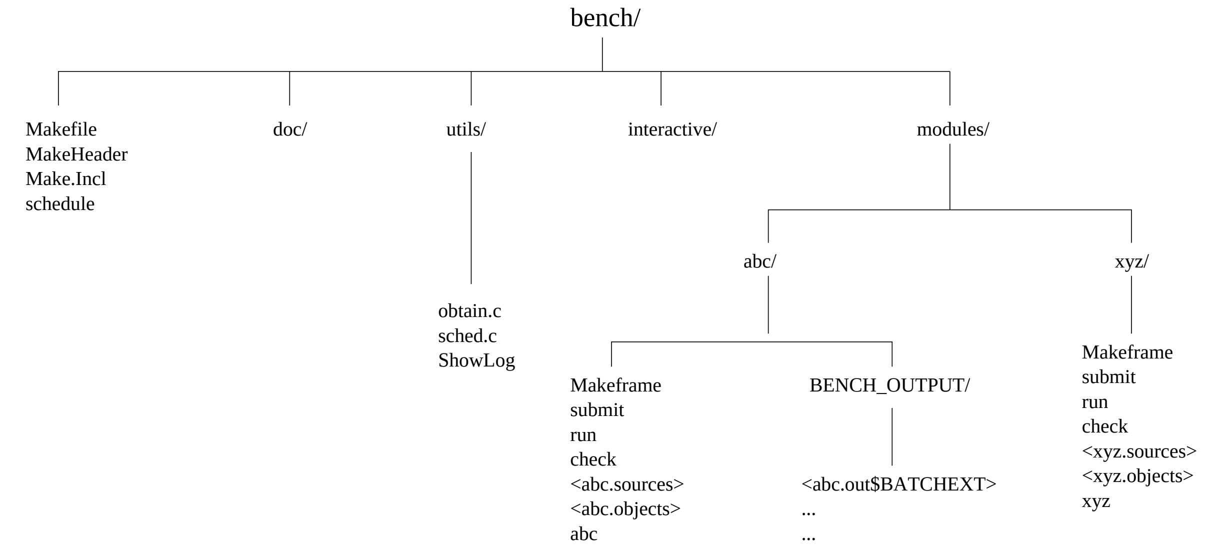

Figure 2: *Directory structure after building modules* `abc` *and* `xyz` *and running module* `abc`.

```
# File scheduler
  0: mod1_run              # run mod1
 10: mod2_run, mod3_run    # run mod2, mod3
 20: 3 * mod1_run          # run mod1, mod1, mod1
```

Figure 3: *Example of a scheduler: at the start,* `mod1` *has to be submitted; after 10 seconds,* `mod2` *and* `mod3` *have to be submitted; after another 10 seconds,* `mod1` *has to be resubmitted thrice. The character "*`#`*" introduces a line of comment.*

3.5.3 Running all modules from the root `Makefile`

This can be done by the command:

```
make run_all BENCHTYPE=<benchtype> BATCHEXT=<batchext>
```

See section 3.5.2 for an explanation of the environment variables. All modules are run just once, in the order specified by the macro `MODIDS` in the top-level `Makefile`.
Note: to guarantee that just one module is active at a time, set up the appropriate NQS configuration and customize the submit script (see section 3.9). Since the throughput benchmark is target is usually run from `schedule`, target `run_all` will in general be used in the context of a single-user benchmark, so the environment variable `BENCHTYPE` usually will be "single".

3.5.4 Running from `schedule`

Thus can be done by the command:

```
make sched CONF=<schedule> BENCHTYPE=<benchtype>
```

Configuration file `schedule` lists a number of modules and, for each module listed, the time at which a module has to be submitted as a batch job. An example of such a configuration file is given in 3. The scheduler is typically used in the context of a throughput benchmark, so the variable `BENCHTYPE` usually will be "thruput".

3.6 Collecting results

All results of a particular run are collected in a separate directory, as determined by environment variable `BENCHID`. The following command

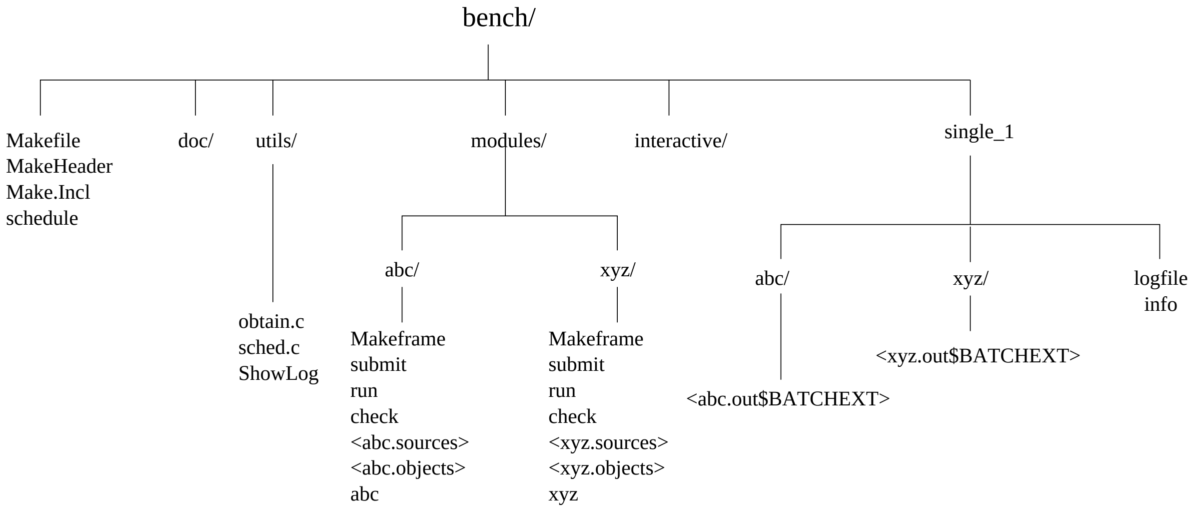

Figure 4: *Directory structure after having run the benchmark named "*`single_1`*".*

```
make collect_all BENCHID=single_1
```

ensures that all results of all modules, residing in the `BENCH_OUTPUT` directories, are moved to a subdirectory of directory `single_1/`. To illustrate this, let us assume that the modules listed in Figure 1 have been built and run. After having issued the above command, the directory structure has become as shown in Figure 4. The collected results in the `single_1/` directory are referred to as second-step results. During the collect step, one is prompted to provide information on the current run. Relevant information would be: the date and time, type of host machine, configuration of the machine, etc. Input can be ended by typing `^D` (Ctrl-D). This information is stored in file `info`.

Analogously to the build and run steps, one can also collect the results of just a single module:

```
make abc_coll BENCHID=single_1
```

3.7 Clean-up

The command

```
make clean
```

removes, for each module, all object files and first-step results, while

```
make clobber
```

removes, for each module, all object files, the executable, the Makefile, and the first-step results. The second-step results can also be removed. For instance, the command

```
make rmres2 BENCHID=single_1
```

ensures that directory `single_1/` will be recursively removed. Notice that the clean-up commands always act on <u>all</u> modules.

3.8 Logfile

For each run (as identifiable through the variable `BATCHEXT`), a file called `logfile` is maintained to which relevant log messages are written. The file provides information about a module's submlssion time and the beginning and end times of its execution. Utility `ShowLog` extracts relevant data from `logfile`, collects the data in a table, and writes this table to standard output. The command to be given is:

```
${BENCHMARKROOT}/utils/ShowLog/<path>/logfile
```

3.9 Setting up the environment

A few things need to be done in order to be able to run a benchmark test on a particular host machine:

- Configure the benchmark root directory.
  The top-level Makefile has to be slightly adapted: the macro BENCHMARKROOT should refer to the benchmark's top directory.
- Configure machine-dependent macros.
  As mentioned, file Make.Incl contains the machine-dependent macros, to be used by each module. One has to ascertain that the macros in Make.Incl apply to the current host.
- Set up the NQS configuration.
- Customize the submit scripts.
  For each module, the submit script has to be adapted to:
  – the host's NQS environment;
  – the module's specific resource requirements;
  – the appropriate NQS batch queue selection for either a "single" or a "thruput" type job submission.
  This is determined by the BENCHTYPE environment variable.
  The NQS qsub command should support at least the following options:
  -x current environment variables are included in the batch job.
  -q <queue> the queue in which the request is to be placed.
  -r <name> the name assigned to tlie request.
  -e <file> the job's standard error file is redirected to file.
  -o <file> the job's standard output file is redirected to file.
  Note: make sure that, in a "single" run, the number of active jobs is limited to just one.
- File space.
  Lastly, and probably superfluously, there should be sufficient file space on the benchmark's filesystem. It should not be problematic to store the results of several runs on this filesystem.

4 Performing the Single User Benchmark

4.1 Preparations

- Create a username bench with Bourne login shell.
- Load the distribution tar file in the homedirectory of user bench.
  – login as user bench.
  – cd.
  – tar -xvf <tar file>.
  (It is not required to use the home directory of bench but all the examples in the documentation assume this).
- Setup the NQS configuration.
  Create a configuration which is capable of running NQS requests one by one sequentially. Create, for instance, a queue "single" with a run limit of 1. Make sure that this queue is capable of satisfying the resource requirements of each module in the single-user benchmark.
- Customize the submit scripts.
  Adjust the submit script in each module directory. The selector BENCHTYPE, which has value "single" in this benchmark, should result in the execution of the appropriate qsub command which guarantees that just one module is executing at a time in this benchmark. Submlt the module, for instance, to the previous mentioned queue "single".
- Define the BENCHMARKROOT variable in the root Makefile.
- Adjust the macro file Make.Incl in the benchmark root directory.
- Generate the module Makefiles in each module directory.
  From the benchmark root directory, give the command:
  make make

- Build all modules and utilities from the benchmark root directory:
  `make build_all`

4.2 Running the Single User Benchmark

- Make sure that no output from previous runs is left hanging around:
  `make rmres1`
- Start system activity data collection.
- Run the modules one by one sequentially:
  `make run_all BENCHTYPE=single BATCHEXT=.0001`
  Notice that the (module-specific) submit scripts will ensure that each request is placed in queue "single".
  Note: To enable valid comparisons with other systems one should be sure that:
  - All modules read, write, and create their files, temporary as well as permanent files, within the same filesystem.
  - No adjustments to the machine's hardware and software are made during this run.
- Collect the results of this run:
  `make collect_all BENCHID=single_1`

  The value of `BENCHID` can be altered (to whatever identifier one prefers). You are prompted to supply information about this run. Specify, for instance, the machine configuration, or other relevant information.
- System activity data during this run.
  Put all the relevant system activity data, usually referred to as "`sar`" data, related to this run in an ASCII readable format in the directory which contains all the data for this run. The system activity data should contain:
  – CPU utilization.
  – Memory utilization.
  – I/O utilization.

5 Performing the Throughput Benchmark

5.1 Preparations

- Setup the NQS configuration.
  Create a configuration which is capable of running multiple NQS requests (in parallel) as efficiently as possible. It is recommended to configure not more than 6 NQS batch queues the modules in the throughput benchmark. An additional pipe queue could also be configured. Again, for a valid comparison to other systems a further restraint on this NQS configuration should be that the CPU priority of each queue, usually referred to as `nice` factor, should be the same for each queue and also equal to the priority of interactive processes.
- Customize the submit scripts.
  Adjust the submit script in each module directory. The selector `BENCHTYPE`, which has value "thruput" in this benchmark, should result in the execution of the appropriate `qsub` command which will eventually place the job in one of the five batch queues.
- Assuming that all module executables exist and have already run in the single user benchmark, perform the steps in the next subsection.

5.2 Running the Throughput Benchmark

- Clean up resuits from previous runs:
  `make rmres1`

  Note: make sure you save benchmark results of previous runs if neccesary.
- Start system activity data collection.

- Run the throughput benchmark from the released scheduling file schedule.
  `make sched CONF=schedule BENCHTYPE=thruput`

  Notes:
  - Once the file `schedule` has been defined one should leave it untouched during the benchmark to enable valid comparison with other systems.
  - All modules should read, write, and create their files, temporary as well as permanent files, within the same filesystem.
  - No adjustments to the machine's hardware and software are to be made during this run.
  - It is recommended to submit the jobs in small groups with small programs to be submitted intermittently in between. One can repeat this stencil several (4–5) times.

- Collect the results of this run.
  `make collect_all BENCHID=thruput_1`

  The value of `BENCHID` can be altered (to whatever identifier one prefers). You are prompted to supply information about this run. Specify, for instance, the machine configtiration, or other relevant information. The first-step results are removed by the collect step.
- System activity data during this run.

  Put all the relevant system activity data, usually referred to as "sar" data, related to this run in an ASCII readable format in the directory which contains all the data for this run. The system activity data should contain:
  – CPU utilization.
  – Memory utilization.
  – I/O utilization.

6 Information about the programs

This section provides information about the only part of the benchmark framework that is actually filled in. It is the EuroBen Benchmark set (V3.9) The necessary information about the EuroBen set, like a short characterization of the programs and a correctness check (if required), etc., is given in this report as it is relatively short and selfcontained. For the part with user programs it is expected that such information is given the `${BENCHMARKROOT}/doc/readme/` directory: program descriptions, correctness checks, contact addresses of contributors, etc. Section 6.2 gives pointers to this information.

6.1 The EuroBen set

The EuroBen set contains FORTRAN77 as well as Fortran 90 programs. All programs in the set are to be run in 64-bit precision. For the FORTRAN77 programs this amounts to `DOUBLE PRECISION` on most systems. For the Fortran 90 programs a `KIND` is selected that ensures this precision. The FORTRAN77 programs are given in a generic form from which the appropriate version can be generated by a simple `make` command as decribed in section 3.4. The precision can be set for machines by `PRECISION=single` for machines with a basic precision of 64 bits, or `PRECISION=double` for machines with a basic precision of 32 bits in the file `Make.Incl` in the `${BENCHMARKROOT}` directory.

6.1.1 General remarks about the programs

In `${BENCHMARKROOT}/modules/esb/` directories named `basics/` and `linalg/` are added to the programs of the modules proper. `basics/` contains six functions/subroutines that have a special position. `basics/` serves two purposes:

1. To provide routines that are common to many of the programs and therefore eases the maintainability of all codes.

2. To isolate the machine dependencies that turn up in the programs.

Care has been taken to write the programs in portable FORTRAN77/Fortran 90, however, some general machine dependencies exist accross the programs and therefore the following machine dependent functions/subroutines are located in basics/:

- The function `secnd()`. It should return the CPU-time in seconds as a 64-bit floating-point value since the start of the program. Please insert into `secnd()` the proper timing call for your local machine.
- The subroutine `state(modnam)` prints some information about the machine, the compiler version, the operating system version and the date and time. Please, bring the information about the machine, the compiler, and the operating system up-to-date.
- The function `wclock()` yields the wallclock time as a 64-bit floating-point value representing wallclock seconds since the start of the program. `wclock()` uses the C function `cclock.c` which in turn calls the system function `gettimeofday()`. On almost any system this gives wallclock times with a few μs resolution and it is usually better than the Fortran 90 routine `system_clock`. If necessary, please customise to get the described functionality for your local system.
- The function `dmacdp(i)` or `smacdp(i)` (depending on the basic precision of the machine) returns a number of machine dependent parameters like the relative precision of the machine, smallest and largest representable floating-point number, etc.

The other routines in `basics/` are subroutine `d/sdummy(a,iconst)` which should prevent over-optimisation in repeating loops and function `d/sran1(idum)` which is a machine-independent uniform random number generator. These two routines need no modification but are put in `basics/` just because they are called in other programs so often. A library containing the routines in `basics/` is generated when one follows the directions in the `README` located in `${BENCHMARKROOT}/modules/esb`. The situation is similar for `linalg/`. This directory contains all relevant BLAS1/2/3 sources and some LINPACK and LAPACK sources in single- and double precision. These sources need no modification. A library is generated from directory `linalg` which will be available to all programs that need it.

With the routines `secnd`, `state`, `wclock`, and `d/smacdp` modified all programs should run without problems, be it that the programs `mod3a` and `mod3b` should be adjusted according to the available main memory size. Please refer to the next subsection for the specific modifications in these programs. As can be gathered frons the information in subsection 6.1.2, many times routines from BLAS, LINPACK/LAPACK, and EISPACK are used. Often optimised library versions of these routines are provided by the vendor and one is encouraged to run the programs that contain them with these optimized versions as well. The same holds for FFT's and the random number generation program that could be compared with the optimized (and machine dependent) versions available from the vendor.

6.1.2 Specific information per file

1. Name: `mod1ac`
   A program to measure the $n_{1/2}$ and the r_∞ of a repertoire of operations +, -, $\times$, /, and important triadic operations like the inner product and the `axpy` operation are performed on consecutive vectors, vectors with odd and even stride and on list-addressed vectors. As for correctness: all $n_{1/2}$ values should at least be positive for the shorter vector lengths (< 100). Because of the simplicity of the model that does not take into account buffers, caches, or swapping one cannot expect more. Fortunately, the $n_{1/2}$ is especially important at small vector lengths while r_∞ is most reliable for long vector lengths.

2. Name: `mod1d`
   A program for the detection of possible memory bank conflicts.

   Correctness check:
   The last line of the output should read:

```
Check (Should be 1000):       1000.0
```

3. Name: `mod1e`
   A program to test the accuracy of the important intrinsic functions.

   Correctness check:
   The program gives a list of deviations from the "exact" values which should be in the order of the relative machine precision.

4. Name: `mod1f`
   A program to test the speed of the important intrinsic functions. An estimate of the $n_{1/2}$ and r_∞ for these functions is given.

5. Name: `mod1g`
   A program to test the cost of memory references relative to arithimetic operations. This is expressed in the value $f_{1/2}$ from evaluating a polynomial of degree 1, ..., 10.
   Remark: This parameter is very sensitive to the resolution of the wallclock and it might produce negative (and thus useless) values on systems with an insufficient timing accuracy.

6. Name: `mod2a`
   A program that measures the speed of matrix-vector multiplies of orders as defined in the input file `mod2a.in`. A correctness check is done internally and reported as a `'T'` value in the result table.

7. Name: `mod2b`
   A program that measures the speed of the solution of dense linear systems of orders as defined in the input file `mod2b.in`. A correctness check is done internally and reported as a `'T'` value in the result table.

8. Name: `mod2c`
   A program that measures the speed of the solution of sparse linear systems of orders $10^3, \ldots, 100^3$. Correctness can be deduced from the residual after 10 iterations for the three implementations considered.

9. Name: `mod2d`
   A program that measures the speed of computing eigenvalues of dense linear systems of orders as defined in the input file `mod2d.in`. A correctness check is done internally and reported as a `'T'` value in the result table.

10. Name: `mod2e`
    A program that measures the speed of computing eigenvalues of sparse linear systems of orders 10, ..., 10000. The correctness is checked against internally available values.

11. Name: `mod2f`
    A program that measures the speed of a 1-D complex-to-complex FFT for orders $2^8, \ldots, 2^{20}$ as defined in the input file `mod2f.in`. A correctness check is done internally and reported as a `'T'` value in the result table.

12. Name: `mod2g`
    A program that measures the speed of a 2-D Haar Wavelet transform for orders 16×16 to 512×256 as defined in the input file `mod2g.in`. A correctness check is done internally and reported as a `'T'` value in the result table.

13. Name: `mod2h`
    A program that measures the speed of generating uniformly distributed random numbers. The correctness is checked against internally available values.

14. Name: `mod3a`
    A program that measures the speed of very large sparse matrix-vector multplications (sizes 20000×25000 to 100000×250000). As the data resides out-of-core, I/O bandwidth is measured. A correctness check is done internally and reported as a `'T'` value in the result table.

15. Name: `mod3b`
An out of core 8,192×8,192 2-D complex FFT. The size of the blocks to be treated in-core and the size of the records used in direct access I/O can be adjusted in the program for optimal performance. Please, change these slzes accordingly.
Remark: For small computers or workstations the machine resources will generally not be sufficient to perform this test. In this case the dimensions of the order may be reduced to 512×512. The following modifications in the source text of `Program mod3b` are required.

    a. Comment the `Parameter` statement at line 80-81 and de-comment the similar statement on lines 82–83.
    b. Comment the assignment statements for `mexa(1)` and `mexa(2)` on lines 105–106 and de-comment the corresponding statements on lines 107-108.
    c. Comment the `Format` statement at line 159 and de-comment the corresponding statement at line 160.

    Please indicate this reduction in size when reporting!
    Correctness check: the check looks like:

```
Correctness check (should be two almost zeros and a one)
   .17373E-14        .21316E-13      1.0000
```

    The almost zero values can vary somewhat due to the difference in floating-point representations.

16. Name: `mod3c`
A zebra-line multigrid code. A Poisson equation on a 257×257 grid is solved. The correctness can easily be inferred from the convergence of the process. After approximately 23 iterations the algorithm stops with a final residual of about `0.531E-10` in the l_2-norm.

17. Name: `mod3d`
A program for the solution of a 500×20 linear least squares problem. The problem is solved two times: once using a QR decomposition and once by Singular Value Decomposition (SVD). The routines from the LLSQ-package of Lawson and Hanson are used. The data for this problem are read from the file `LSQDAT`.

    Correctness check:
    Correctness can be inferred from two lines under the headers
    `Part 1:  QR decomposition` and `Part 2:  SVD decomposition`:
    The values in these lines should be identical and should read:

```
Pseudorank of A        20
Norm of residual       .986E+00
```

    and

```
Rank of A              20
Norm of residual       .986E+00
```

    respectively. Also the solutions given by both methods should be identical (or should differ very slightly). The absolute difference of both solutions is printed and should be in the order of the machine precision.

18. Name: `mod3e`
`mod3e` solves a non-linear least squares problem with 7 fitting parameters and 1,000 observations. Standard MINPACK routines are used. These in turn heavily use the BLAS routine `D/SNRM2`. This routine is included in the `linalg` library. If an optimised version of `D/SNRM2` exists on your machine please substitute it for the `linalg` version to assess the effectiveness of this optimisation.

    Correctness check:

```
Parameter values at optimum are:
  997.07          11.994       263.59        1.1009        72.500
   .21692         42.650
Ideal parameter values (without 15% noise) should be:
  1000.0          10.000       300.00        1.0000        75.000
  .20000          50.000
```

19. Name: `mod3f`
Program that solves a one-dimensional diffusion (desorption) problem. Solution is done based on the routine `LSODE` from ODEPACK. For the solution of the arising linear systems the LINPACK routines `D/SGEFA` and `D/SGESL` are used. These routines are located in the library `linalg`. References to `D/SGBFA` and `D/SGBSL` are made as well, but for the problem considered here this branch of the program is not taken. When optimized versions of BLAS or LINPACK routines are available, please, also run this optimised version to assess the effectivity of the optimization.

Correctness check: the following lines are printed:

```
first component:       .4615E+06
last component:        .4621E+06
```

20. Name: `mod3g`
`mod3g` solves Poisson and Helmholtz equations by cyclic reduction and Fast Fourier techniques (i.e., a "Fast Poisson Solver"). The Laplace equation on the unit square is solved with boundary conditions $U = 1.0$ on the entire boundary ∂G. The discretisation is on a 257×257 grid.

Correctness check: the important check lines are:

```
Max.abs.error =      .306E-11   at .50, .50
Max.rel.error =      .306E-11   at .50, .50
```

21. Name: `mod3h`
Program that solves a Poisson equation by a block-relaxation method. The problem sizes range from 17×17 to 513×513.

Correctness check: the important check lines are:

```
 Nx    Ny  Max. it   Iter    Ucheck      Residual
-----------------------------------------------------
  17    17      300    234      1.00       0.11082E-13
  33    33      300    300     0.993       0.43737E-05
  41    41      300    300     0.953       0.13114E-03
  65    65      300    300     0.605       0.36537E-02
 129   129      300    300     0.345E-01   0.10004E-01
 257   257      300    300     0.712E-06   0.19677E-01
 513   513      300    300     0.132E-23   0.39277E-01
```

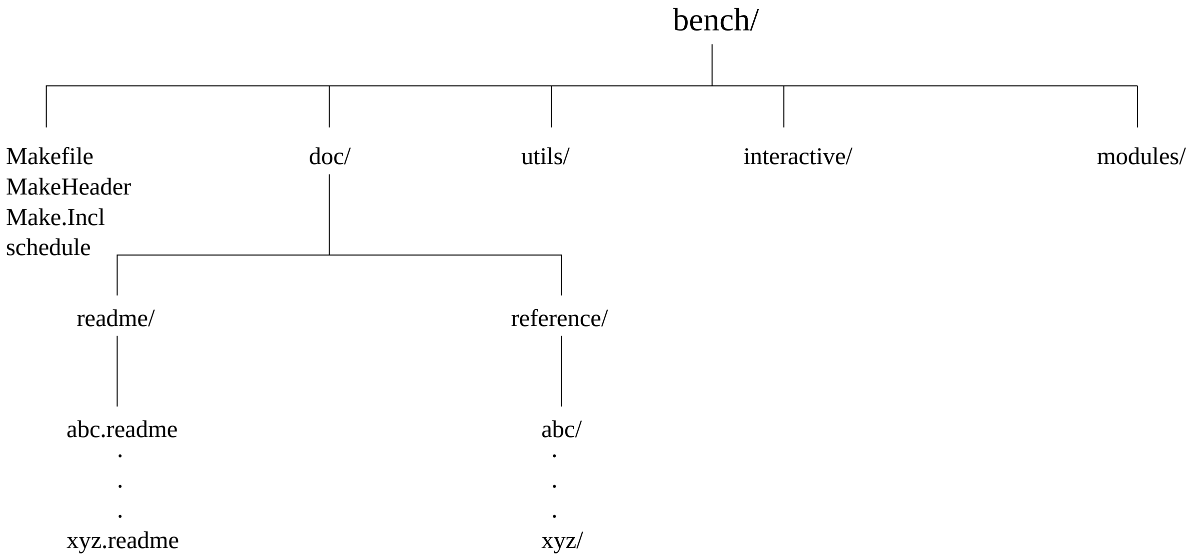

Figure 5: *The location of the* README *files containing ihe information regarding the user programs in subdirectory* readme/ *in directory* doc/. *The* reference/ *subdirectory in* doc/ *has subdirectories which contains information used in correctness checks.*

6.2 The user programs

The information about the user programs is expected to be located in a set of README files as shown in Figure 5.

Directory ${BENCHMARKROOT}/doc/ is meant to contain all information that is required for running and checking the programs. This information is divided over the two subdirectories readme and reference/. Subdirectory readme/ should contain README files <prog>.readme, one for each program <prog>. Each README file in turn should contain:

- **Name**: The name of the program.
- **Origin**: The name, address and other information about the contributor of the program.
- **Description**: Short description of the program.
- **Resource requirements**: Estimate of central memory usage, wallclock time for execution, and the maximal file space required.
- **Correctness**: An explanation how correctness of the program can be checked. Note, however, that this in most cases will be done by the check scripts mentioned in chapter 3.
- **Person to contact**: Information about the person that may be contacted when difficulties with the program arise. Usually, address, telephone number, fax number, and email address are given.
- **Structure of the program(s) and data**: Here the relevant source names, input files and output files are given. If there is a sequence of dependent programs, their dependencies are described.
- **Dependency on libraries**: Description of dependency of libraries like BLAS, (SCA)LAPACK, NAG, etc.
- **Portability of Fortran (C) source**: Notes on non-standard, and potentially non-portable, features of the program and what actions have been taken to remove them.
- **Scaling**: Note on the scaling aspects of the code, mostly in terms of mesh sizes, number of time steps, number of iterations, etc.

As already remarked, subdirectory reference/ in turn contains subdirectories for each program with the necessary files to check the correctness of the programs. The structure of these subdirectories is usually so diverse that a description here is not sensible. The correctness information in the README files, together with the files in the subdirectories of reference/ should be sufficient to assess the correctness of the programs.

7 The interactive benchmark

7.1 Introduction

The set of programs and shell scripts described in this section constitutes a benchmark suite for measuring the interactive performance of a target UNIX system using one or more front-end workstations. Typically, this target system is a large, shared system, such as a number cruncher or file server.

Unfortunately, a good definition of "interactive performance" is hard to give. Assuming that it is impossible to express such a complex notion by a single number, we propose a rather liberal definition, and define interactive performance as the overall responsiveness of a target system under various conditions. In this context, we define "reponse time" as the time difference between the transmission of a command to the target system and the receipt of the target system's response.

The benchmark suite is based on the idea, that every workstation runs a benchmark script that repeatedly measures the responsiveness of the target system. The benchmark suite consists of the following components:

- a script interpreter called `tjeg`.
- a number of *session script files*.
- a number of *shell scripts* and *utility programs*.

The script interpreter constitutes the core of the suite. The interpreter drives the terminal side of an interactive shell, providing a means to execute `telnet`, `ftp` or other network application sessions on the basis of a session script file. The interpreter connects to an interactive shell via a pseudo-terminal or *pty*.

The session script files describe a series of commands to or through the network application in an application-independent way. The script files in this "session language" are legible text files and can be easily modified with a text editor, such as `vi`.

The interpreter and session script files are accompanied by several shell scripts and utility programs performing installation, control, and analysis tasks.

The benchmark suite measures the response times of UNIX commands. The system's response is matched against a specified regular expression. Measured response times are logged in legible text files, suitable for later processing to obtain statistics. The target system's response may be written to a "capture file" that is specified in the session script file.

There are several possibilities to use the benchmark suite, ranging from a single script initiated from a single workstation to multiple scripts that are initiated from multiple workstations. The latter version can be used to simulate a "representative" interactive workload.

The benchmark suite was developed on a Sun Sparcstation 1, with 327 Mbyte hard disk, 150 Mbyte tape cartridge (DC600A compatible), 12 Mbyte of main memory, and running Sun05 4.0.3. The suite was ported to various other machines running their own Unix flavor e.g. DEC Vax (BSD 4.3), IBM 6151 (AIX/RT) , IBM 3090 (AIX/370), Apollo DN10000 (Domain/OS release 10.2), Gould NP1 (UTX 32 release 3.1), SGI Indy (IRIX 6.3), and SGI Origin200 (IRIX 6.4).

The next section describes the required configuration and all operational aspects of the benchmark suite. Section 7.3 describes the operation of the `tjeg` interpreter and gives a description of the script language. Section 7.4 focuses on the benchmark script files.

The standard interactive benchmark runs in a multi-workstation environment. It may be possible that the steps described in this part of the document can be followed without problems. lt is likely, however, that there will be situations where it is not possible to completely follow this Benchmark Guide. Trouble-free installation and operation of this suite cannot be guaranteed because of the abundancy of possible configurations and different UNIX flavors. In case of operational errors, modification of the benchmark software, the script files, or even the entire procedure may be unavoidable. Full knowledge of the internal operation is a prerequisite to make proper modifications. Therefore, it is advised to read this entire Technical Documentation before applying the steps described in this guide.

7.2 Usage

7.2.1 Distribution

The interactive benchmark is distributed both as a part of a gzipped `tar` file that also contains the EuroBen Throughput Benchmark Framework and as a separate gzipped `tar` file. It can be downloaded in either from from `www.phys.uu.nl/~steen/programs`.

7.2.2 Required configuration

As already mentioned, the benchmark software runs on one or more workstations. Specifically, the configuration must consist of the following components:

1. One or more UNIX workstations that are used to execute the benchmark software. The workstations should have access to at least 10 Mbytes of disk space; a local hard disk for each workstation is preferable. The system must support the so-called pseudo-terminal or `pty` interface (`/dev/ptyxx`).
2. A TCP/IP network that connects the workstations to the target system.
3. Means to save the benchmark results offline.
4. Access to a printer to make hard copies of analyzed performance data.

7.2.3 Installation preparation

As already mentioned, it is possible to run the benchmark in a multi-workstation environment. The procedure that is followed to install the software depends on the actual configuration. The following configurations are distinguished:

1. single workstation, single user.
   This is the most basic configuration. A single benchmark is run under a single username on a single workstation. In this configuration, the benchmark suite can be installed in one pass.
2. multiple workstations, multiple users.
   This is a more advanced configuration. Multiple benchmarks are run on multiple workstations. Each workstation has its own user with his own file system. You will have to perform as many installations as there are workstations; the installation procedure must be performed on every workstation from which the benchmark will be initiated.
3. multiple workstations, single user.
   This advanced configuration runs Sun Yellow Pages (YP) and Sun Network File System (NFS). Multiple benchmarks run on multiple workstations, but under a single username. This setup is characterized by a central user validation and a distributed file system. Although there are several workstations, there is only one user and only one file system. Therefore, it is sufficient to perform only one installation.

User validation

In a single workstation environment (1), you must administrate and validate a non-priviliged user on the workstation. In a multi-workstation, multi-user environment (2), you must administrate a non-priviliged user on every workstation. In a multi-workstation, single-user environment (3), you must define one non-priviliged user and permit this user to access all workstations involved. In all cases, the Bourne shell (`/bin/sh`) must be defined as the above users' default shell.

Note:
In fact, the interactive benchmark assumes `sh` or `ksh` to be used on the workstation used to initiate the test. It will *not* work with the `csh` or similar shells. However, a `csh`-like might be used on the target system (i.e., the system from which we want to measure the interactive performance). If the directions in the manual have been followed in the `setup` phase, the benchmark should be able to recognize a `csh` prompt and, accordingly, start a `csh`.

The benchmark suite also requires a non-privileged user on the target system!

Installation steps (1–4)

This paragraph describes the 4 steps that must be followed to complete the installation process. It is assumed that the benchmark suite is installed from scratch.

Note: In the text below, type-ins are displayed in a *sloped courier* font.

STEP 1 – Login on workstation and remove existing files

Log in to the workstation under the user name that will be used to conduct the benchmark.

Since the benchmark suite is installed from scratch, remove all existing files and directories from this user's home directory:

*rm -rf *; rm -rf . **

STEP 2 – Download files and directories

In this step, the benchmark suite's files and directories are downloaded from the fileserver that holds the benchmark:

- *ftp <fileserver>*.
- On the fileserver: *cd interactive*
- *prompt off*
- *mget **
- *quit*

On the workstation we have now the following files:

`Makefile README bench.tar.uue ibig.c`

Copy the files `Makefile`, `README` , and `ibig.c` to the target system. This `Makefile` is used to compile program `ibig.c` on the target system, resulting in four different files. Note that this is *not* the same `Makefile` that is stored in the top directory of `bench.tar` below! Also note that it is not necessary to compile the benchmark by hand: it is one of the commands in the interactive benchmark.

Unpack the `uuencoded` `tar` file by typing:

uudecode bench.tar.uue; tar xvf bench.tar

After this step, the user's home directory contains a file `AAAREADME`, a new `.profile`, and the following directories:

`bin`:	– shell scripts for installation, configuration and operation.
`logfiles`:	– contains the benchmark results: log files & capture files.
`sessions`:	– contains the session script files.
`sessions/include`:	– contains files that are included by the session scripts.
`shellscripts`:	– contains supporting shell scripts.
`src`:	– contains C-sources of the benchmark program (`tjeg`).
`utilities`:	– contains C-sources of supporting programs.

STEP 3 – Build the benchmark software

In this step, you compile the benchmark software.

Important: to activate the new profile, you must first logout: `^D`

logout

and login again. Subsequently, set the `TERM` variable to match the terminal you are working on:

TERM=<terminal type>

export TERM

Next, compile the benchmark sources and build the benchmark environment by typing:

install

During the build process, several messages will be displayed. The last message is `... Build Completed ...` This message, however, does not guarantee that the build process has been successful. Be attentive to compiler error messages!

Although the build process has been tested on workstations of different vendors and types, a flawless installation cannot be guaranteed. If this step yields any errors, you will probably have to apply some modifications to correct the problem, and repeat this step.

STEP 4 - Initialize the benchmark environment

After the build process (step 3) has successfully completed, you'll have to specify a number of things to be able to successfully login on the target system. To accomplish this, run the `setup` shell script.
Type:

setup

The setup script prompts you to answer the following questions:

- What is the IP-address/domain name of the target system?
- What is the username on the target system to run the benchmark under?
- What is password of the user on the target system?
- What is the prompt of the Bourne-shell on the target system?
- What is the prompt of the c-shell on the target system?
- What is the IP-address/domain name of this workstation?
- What is the username on the workstation to conduct the benchmark?
- What is the password of the user on the workstation?

Note:
If you supply an incorrect answer to any of these questions, the benchmark suite will not function properly!

After compieting step 4, the benchmark suite is ready for operation. Since all supplied information is stored on disk, the workstation will remain benchmark-ready even if it is switched off and on.

Important:
Note that, if you are running the benchmark in a multi-workstation, multi-user environment, you must repeat steps 1-4 for every workstation/user in your configuration!

7.2.4 Operation

This paragraph describes how to initiate (and terminate) a script file on one workstation. A script file is not a shell script, but a small program in a special language that is interpreted by the script interpreter (`tjeg`). In essence, the script file defines how the script interpreter communicaties with the outside world. See section 7.3 for more detailed information. Script files reside in the directory sessions, and have suffix `.scr`.

Typically, the script file will be the benchmark script, i.e., the script that is used to measure the interactive performance of the target system. All relevant script files are described in subsection 7.2.5 and, in more detail, in section 7.4.

The execution of a script file is controlled by two commands: *start* and *stop*. The start command allows you to initiate one or more scripts on one workstation. Each script is identified by a unique session identification number. The `stop` command is used to terminate all scripts that were initiated by the same `start`.

`start`
This shell script can be invoked in three ways:

1. Without an argument; it will present a help text similar to this text. Type:

   start

2. With one argument `[scriptfile]`. Type:

   start [scriptfile]

   This will cause the script to be run once in the foreground. This option can be used to test whether the script works properly.

   `[scriptfile]` is part of the name of a script file `[scriptfile].scr`, residing in the directory `sessions/`. This script is interpreted by the benchmark program (`tjeg`).

   The results of the session are appended to `logfiles/[scriptfile].[session-id].log`, where `session-id` is a unique number associated with each copy of the session script.

3. With a pair of arguments `[scriptfile] [Iogfile]` and optional parameter `[n]`:

   start [scriptfile logfile [n]]

   where n is an integer value. This will cause n instances of the script to be run over and over as background processes until they are stopped by the `stop` command. These instances are initiated with an interval time of $180/n$ seconds. The default repetition time for each instance is 1 minute; these intervals are defined in the shell script `run`.

   `[scriptfile]` is part of the name of a script file `[scriptfile].scr`, residing in the directory `sessions/`. This script is interpreted by the benchmark program (`tjeg`).

   The results of the session are appended to `logfiles/[scriptfile].[session-id].log`, where `session-id` is a unique number associated with each copy of the session script.

   Regardless of the number of arguments, start can be called with additional flags:

   - `-m` runs the script in monitor mode; all data sent and received is displayed on the terminal screen. Because this mode may influence the performance data, log data is written to `logfiles/[logfile]-m.[session-id].log`.
   - `-v` runs in debug mode. Depending on the value of n, more or less detailed information of the internal information is displayed.
   - `-s` runs in silent mode (all output is suppressed).

   These flags must be specified immediately after the start command. For example:

   start -m script

   Additional information about the flags is given in the next section.

   For each script that is initiated, `start` will append an entry to the history file `logfiles/history`.

`stop`

Script files running in the background can be stopped with the stop command.

stop [scriptfile]

This will cause all instances of the script file called with `start [scriptfile]` to stop when the current iteration has ended. This may take some time! A message will be displayed for each script instance that is terminated. For each script that is terminated, stop will also append an entry to the history file `logfiles/history`.

Important
If you are running scripts in a multiple-workstation, single-user Sun YP and NFS environment, type:

. sunclient

before starting any scripts.

Important
If you encounter an error during script execution, stop the script using the `stop` command. A failing script may leave dangling processes. lt is advised to check for such processes by typing:

ps

If there are any dangling processes, kill them using the kill command. Always stop scripts using `stop` before killing processes!

7.2.5 Script files

The benchmark suite contains three predefined session scripts: `bench.scr`, `mark.scr`, and `station.scr`. Section 7.4 describes these scripts in some more detail.

`bench.scr`

The main script, `bench.scr`, measures the interactive performance of the target system. For this purpose, the script opens a `telnet` connection to the target system, and measures the response times of a collection of remote UNIX commands.

If you run the benchmark in a multi-workstation, single-user Yellow Pages and NFS environment, type:

. sunclient

before starting this script file.

It is possible to run multiple, concurrent instances of this script on the same workstation. In general, n instances of this script running in the background are invoked by typing:

start bench [logfile] n

Consequently, the response times measured by these instances are logged in n files named `sessions/[logfile].[session-id].log`, where `session-id` is the unique session identification number assigned to each instance.

Running concurrent instances of this script is safe. Internally, the script uses the session identification number to avoid interference with other instances of the same script. Running too many instances, however, may influence the performance of the workstation, which may afflict the accuracy and representativity of the measurements. Do not run more than 5–10 benchmark scripts in parallel.

Estimated elapse time: 2–10 minutes per iteration.

`mark.scr`

This script is identical to `bench.scr`, except that several bottleneck-prone activities have been removed from the original script. This script can be used to simulate an interactive workload from a different workstation.

If you run the benchmark in a multi-workstation, single-user Yellow Pages and NFS environment, type:

. sunclient

before starting this script file.

To invoke n instances of this script, type:

start mark [logfile] n

Estimated elapse time: 2–10 minutes per iteration.

`station.scr`

This small script is used to determine several characteristics of the front-end workstation. Type:

start station

The script will produce a file `logfiles/station.[session-id].log`, which contains the relevant information.

Estimated elapse time: 1 minute per iteration.

See section 7.4 for more information on these scripts.

7.2.6 Data collection and analysis

After compietion of a sequence of benchmark scripts, the workstation that has been used to run the benchmark scripts contains a directory `logfiles` that is filled with log files. Specifically, this directory contains a log file for each script instance.

To produce a quick analysis of the measurement results, it is required that these separate logfiles are collected into one logfile. The benchmark suite contains three shell scripts for data collection and analysis: `collect`, `analyse`, and `graph`.

<u>`collect`</u>

This shell script gathers the logfiles and capture files of all instances of the script that were started with the same `logfile` argument. These logfiles and capture files are named `logfiles/[logfile].[session-id].log` and `logfiles/[logfile].[session-id].log.cap`, respectively. The shell script produces two files: `logfiles/[logfile].log` and `logfiles/[logfile].log.cap`. Type:

`collect [logfile]`

Where `[logfile]` is identical to the `logfile` argument that was passed to the `start` command.

This operation is destructive, meaning that the logfile and capture file of each separate instance are destroyed when the collect operation completes.

<u>`analyse`</u>

If the logfiles have been succesfully collected, you can produce a draft analysis of the benchmark results using the shell script analyse. Type:

`analyse [logfile]`

Where `[logfile]` is identical to the `logfile` argument that was passed to the `collect` command. Analysed logfile data is written to the terminal screen in the following format:

`<frequency> <mean response time> <standard deviation > <command>`

The meaning of these fields is as follows:

`frequency`:	– the number of times this particular command was found in the logfile.
`mean response time`:	– the command's mean response time in milliseconds, calculated over all its entries in the logfile.
`standard deviation`:	– the standard deviation in milliseconds of these entries.
`command`:	– the command of which the responsetime was measured.

<u>`graph`</u>

This shell script produces a sirnple bar-graph of the response time distribution of command entries in the logfile. Type:

`graph [logfile] [scale size]`

The scale size should be expressed in milliseconds and corresponds to the maximum number of asterisks that can be displayed in a single row (currently defined as 60). The default value is 1000 milliseconds.

Each command entry in the graph has the following format:

```
command: <command>
 <time stampl>        <response time>    ******
 <time stamp2>        <response time>    *********************
      ...
 <time stampn>        <response time>    *************
```

Important
If you are running the benchmark in a multi-workstation, multi-user environment, there is log data to collect, analyze and display for each workstation/user!

7.2.7 Saving the results

After collecting, analyzing and displaying the data, it is time to save the benchmark results. To this end a command *save* is available.

save

This command automatically tars and uuencodes the entire directory logfiles and transfers the uuencoded tar file to the target system using ftp. Type:

save [file] [target user password]

By default, this command transfers the specified file to the home directory of the user that was specified with the setup command. You can specify an alternate host/username/password on the command line.

Note:
be careful when you specify the file name. If a file of the same name already exists in the home directory of the user on the target system, it will be overwritten!

Important
If you are running the benchmark in a multi-workstation, multi-user environment, you must save the log files for every workstation/user in your configuration. Be careful not to overwrite the log data of other workstations!

The collected results should be put on the same storage medium that also contains the results of the single-user and throughput benchmarks to obtain a complete survey of the target system's performance.

7.3 The tjeg interpreter

7.3.1 Introduction

Just like its name, tjeg is a language which is small and ugly. lt provides a way to describe an interactive session on a UNIX system and to measure the response times of single or multiple UNIX commands.

A tjeg script consists of a mix of tjeg language and input for the session. All tjeg commands start in the first column of a line and begin with a # character. All other lines are input for the session.

tjeg consists of two passes. The first pass reads the tjeg script and builds an internal version of it in memory. The second pass executes the tjeg commands from the internal version. This execution pass starts a shell which will receive all session input. The output this produces is processed by tjeg.

The tjeg parser is fairly simple. It does not perform extensive error handling. A few simple rules the parser uses:

- tjeg commands start in colomn 1.
- The first parameter is separated from the command by 1 space.
- If tjeg does not recognise a line it assumes it is session input.

If you see tjeg sending lines to the session that look like tjeg commands, tjeg encountered an illegal statement and assumed it was session input.

Calling tjeg

Synopsis: tjeg [-x] [-vn] [-m] [scriptfile]

-x: – do not execute the script, only parse the script.
-vn: – debug level. A higher value for n generaties more debug information.

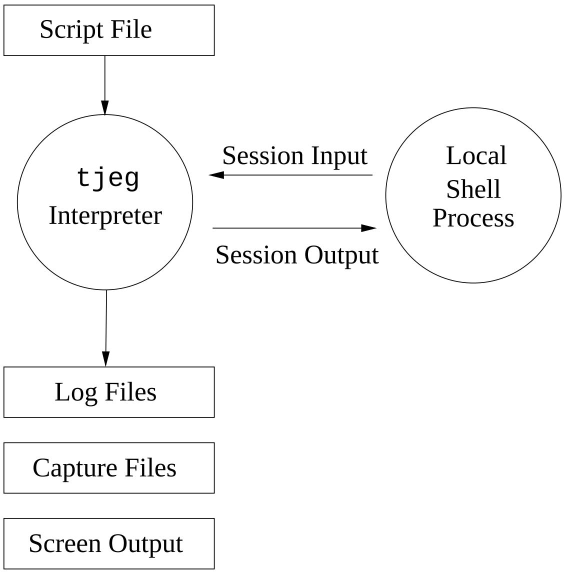

Figure 6: `tjeg` *operation.*

1 show execution.
2 also show parsing.
3 also show internal version of script.
> 3 also show information interesting for `tjeg` programmers only.

`-m`: – monitor communication during the session. Every time something is sent or received from the session `tjeg` displays the data. Every buffer shown is preceded by a bytecount. A '>' at the start of a line denotes data sent to the session, a '<' denotes data received from the session.

`-s`: – `tjeg` runs in silent mode. This means that all messages generated by `#MSG` commands are suppressed. This option has no influence on communication monitoring (`-m` option).

The operation of `tjeg` is depicted in figure 6.

`tjeg` maintains 26 timers (A through Z). These timers can be used to measure the time elapsed between session input and the output the session produces. Furthermore, timervalues can be written to the screen or a log file.

Some of the parameters of the tjeg commands can be derived from shell environment variables. These variables must exist before you activate `tjeg` and are evaluated during the parsing pass of `tjeg`. You can refer to an environment variable by specifying `$(variable-name)`. Refer to the end of this section for more information.

7.3.2 `tjeg` commands

Parameters that are expanded (using environment variables, interpreting quotes and octal characters) are marked with a *.

`#START timer stop-string [MAX max-time label]`

Parameters:

	`timer`	Name of the timer to use. (A through Z).
*	`stop-string`	String that will stop `#END` suspension.
*	`max-time`	Maximum time in seconds that `#END` will suspend.
	`label`	On a timeout `tjeg` will continue at this label.

Example: `#START A "$.*" MAX 10 Timed_Out`

This command starts the timer and sets the stop-string. If a maximum time is specified (in seconds) it also sets a timeout timer, otherwise no timeout will occur. A timeout results in a jump to the label `Time_Out`. Normally this command is followed by some session input, followed by a`#END` command. Note that `tjeg` execution continues after this command. (unlike the `#SYNC` command.

`#END`

This command suspends script execution until a match with the stop-string is found in the session output or a timeout occurred. If a timeout occurred `tjeg` jumps to the timeout label.

`#LABEL label`

Parameters:

	`label`	label name.

Example: `#LABEL Timed_Out`

`#GOTO label`

Parameters:

	`label`	label name.

Example: `#GOTO Timed_Out`

`#SYNC stop-string [MAX max-time label]`

Parameters:

*	`stop-string`	String that will stop `#SYNC` suspension.
*	`max-time`	Maximum time in seconds that `#SYNC` will suspend.
	`label`	On a timeout `tjeg` will continue at this label.

Example: `#SYNC "$.*" MAX 15 Timed_Out`

`#SYNC` is different from `#START`. First of all it does not maintain a timer. Furthermore, `tjeg` execution is immediately suspended waiting for the stop-string to occur in the session output. If a string matching the stop-string is not received before 15 seconds have elapsed, execution continues at the specified label. Note: a sequence like:

```
echo "tjeg"
#SYNC "tjeg.*"
```

may not work since the echo output is received before the `#SYNC` command is executed. Instead of this `#START` should be used, for example:

```
#START "tjeg.*"
echo "tjeg"
#END
```

`#WAIT wait-time`

Parameters:

*	`wait-time`	Time to wait in seconds.

Example: `#WAIT 10`

`tjeg` execution is suspended for the specified number of seconds. Session output which is received during this time is skipped.

`#EXIT`

Example: `#EXIT`

Stop `tjeg` execution. The session process is killed and `tjeg` ends.

`#SEND line`

Parameters:

*	`line`	Data which must be sent as session input.

Example: `#SEND telnet $(REMOTE-HOST)`

`#WRITE timer string`

Parameters:

	`timer`	Timer name (A through Z).
*	`string`	Comment text.

Example: `#WRITE A Time to execute command $(SESSION-COMMAND)`

If the logfile is active, a line is written to the logfile containing: `yymmdd hhmmss xxxx string`, where
– `yymmdd` represents the date,
– `hhmmss` represents the time and
– `xxxx` is the timer value in milliseconds.

The timestamp is the time at which the specifiek timer was stopped, not the time at which the `#WRITE` command is executed. If the timer was not stopped yet, the time elapsed since the start of the timer will be shown as a negative number.

`#MSG timer string`

Parameters:

	`timer`	Timer name (A through Z).
*	`string`	Comment text.

Example: `#MSG A Time to execute command $(SESSION-COMMAND)`

If the logfile is active, a line is written to the logfile containing: `yymmdd hhmmss xxxx string`, where
– `yymmdd` represents the date,
– `hhmmss` represents the time and
– `xxxx` is the timer value in milliseconds.

If the specified timer was Z, only the string is written. This enables you to generate terminal messages from a `tjeg` script. The timestamp is the time at which the specifiek timer was stopped, not the time at which the `#MSG` command is executed. If the timer was not stopped yet, the time elapsed since the start of the timer will be shown as a negative number.

`#LOG [filename]` Parameters:

`filename`	Filename of new logfile.

Example: `#LOG tjeg.log`
`#LOG`

If no filename was specified, the current logfile (if any) is closed and subsequent `#WRITE` commands have no effect. If a filename was specified, subsequent `#WRITE` commands are appended to the specified file.

`#CAPTURE [filename]` Parameters:

`filename`	Filename of new capturefile.

Example: `#CAPTURE tjeg.log`
`#CAPTURE`

If no filename was specified, the current capturefile (if any) is closed. If a filename was specified, subsequent session output is appended to the specified file. Remember that if you are using such a capture file, response times you measure may be influenced by the I/O this generates on the machine executing `tjeg`.

`#!local-command` Parameters:

`local-command`	A shell command.

Example: `#!rm -f tjeg.log`

During `tjeg` execution the command is executed by a subshell on the workstation.
Note: The exclamation mark must be followed by at least one space.

`#TIMER ON|OFF timer` Parameters:

`ON` or `OFF`	Start or stop the specified timer.
timer	The name of the timer.

Example: `#TIMER ON A`
`#TIMER OFF A`

`##comment`

Example: `## This is a comment`

`#INCLUDE filename` Parameters:

*	`filename`	File to include in `tjeg` script.

Example: `#TIMER tjeg.inc`

During the parsing pass of `tjeg` all lines of the specified file are included into the `tjeg` script. `#INCLUDE`s can not be nested. After all lines are included, the parser resumes on the line following the `#INCLUDE` command.

`#REPEAT count` Parameters:

*	`count`	Repeat count.

Example: `#REPEAT 100`

This is the start of a loop. Loops can be nested, but you must take care not to jump out of a loop. If you do so, the next `#REPEND` `tjeg` encounters, will (if all iterations were not done yet) continue with the next iteration. Remember `tjeg` is an interpreter, not a compiler so constructs like

```
#REPEAT 5
 ...
#GOTO Label
 ...
#REPEND
 ...
#GOTO Done
#LABEL Label
 ...
#REPEND
 ...
#LABEL Done
```

are valid, although not very clear. The two `#REPEND` commands in this example both belong to the `#REPEAT` command.

`#REPEND`

Example: `#REPEND`

This indicates the end of a loop. If all iterations were done `tjeg` continues with the next command, if not `tjeg` jumps back to the beginning of the loop. A `#REPEND` command jumps back to the last `#REPEAT` command whose loop is not finished.

Parameter expansion

If a parameter is expanded during the parsing pass of `tjeg`, the following rules are applied to the parameter.

`\xxx`	(where `xxx` is an octal number): expands to one character with the specified ASCII value. You must use three octal digits.
`\c`	(where `c` is any character): expands to one character `c`. This is handy to escape the \. E.g. \\ expands to one \.
`$(variable)`	is expanded to the value of the environment variable. If there is no such variable this is a fatal error.
`"text"`	text within double quotes is seen as one parameter, so this is the way to include spaces and tabs in a parameter. You can use `$(variable)` references within such a string.
`'text'`	text within single quotes has the same meaning as above, but `$(variable)` references are not evaluated.

Examples:
Suppose environment variable XYZ has value "hellothere".

`#SEND \004` sends a `^D` to the session.
`#SEND \$` sends a `$` to the session.
`#SEND $(XYZ)` sends `hellothere` to the session.
`#SEND " '$(XYZ) "` sends `'hellothere` to the session.
`#SEND ' "$(XYZ) '` sends `"$(XYZ)` to the session.

7.3.3 Final notes and cautions

An alternative version of `tjeg` can be built and installed by typing:

```
cd
cd src
make spare
cd
```

This version does not use the `pty` pseudo-terminal interface. Instead of this, UNIX-sockets are used as communication channels to the shell process.

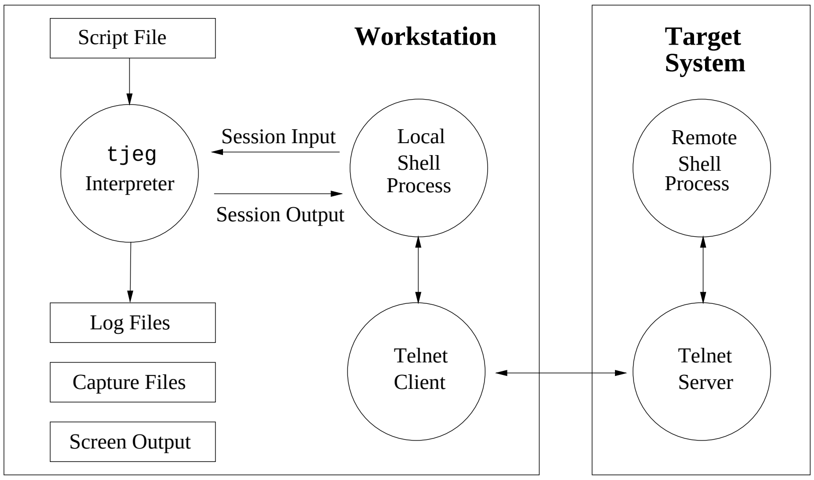

Figure 7: *Operation of* `bench.scr`*.*

Note:
This spare version of `tjeg` is not tested extensively and should be used only as a last resort.

7.4 Script files

The benchmark suite contains a number of script files, which are interpreted by the `tjeg` interpreter running on the workstation. After completing the installation and build phases (steps 1, 2, 3, and 4 in section 7.2.3), the home directory of the user on the workstation contains a directory `sessions`. This directory contains at least the following files and directories:

- `bench.scr`
- `mark.scr`
- `station.scr`
- `include/`

`include/` is a directory containing files that are included from the main script; the script files are described below. The script files can be initiated by the `start` command described in section 7.2 (do not forget to execute the `setup` command first!).

<u>`bench.scr`</u>

`bench.scr` is the main script. During a normal run, the script `bench.scr` is repeatedly interpreted by the `tjeg` interpreter. The interpreter parses and interpretes the main script, and passes the commands encountered in the script file via a pseudo-terminal, or `pty`, to a local command shell for execution; after execution of the command, the response of the local shell is passed back to the interpreter. The interpreter has timing facilities to keep track of the time elapsed between passing the command to the local shell and receiving its response. See the previous section for detailed information.

The operation of `bench.scr` is graphically represented in figure 7.

In essence, the script file imitates an ordinary interactive user, who logs in on the target system via`telne` or `rlogin`. Therefore, the benchmark script orders the local command shell to open a telnet or rlogin connection to the target host. The effect of this `telnet` connection is that the script operates on a `telnet` client rather than to a local shell; the telnet client connects to a telnet server, whi ch connects to a remote shell; this remote shell is, of course, a shell running on the target system.

If the telnet connection has been successfully set up, the script issues a carefully selected set of UNIX commands. The response times measured are the response times of the target host, and thus correspond to the response times an ordinary interactive `telnet` user would experience. Note that the measured response times include network communication overhead! No attempt has been made to eliminate network overhead from the benchmark results. After all, communication time is a determinant factor in system responsiveness.

The main script does little more than including files from the include directory. The include files actually do all the work. This work can be divided in three phases. In the initial phase, the script logs in and prepares its environment. In the second phase, the script performs the benchmark. In the last phase, the script cleans up its environment and logs out.

First, the script tries to connect to the target system. This purpose requires the execution of three script files (assumming we are in `$HOME/sessions/include`). The first include file is:

    ping.i

which issues a `ping` comrnand to the local shell. `ping` is used to check if the target system is alive. If it is not, the script is aborted. The `ping.i` script is excluded by default, since not every UNIX flavor supports this command in the same way. You may include the ping file by de-commenting the line `##INCLUDE include/ping.i` in the file `bench.scr`.

The next include-file called by bench.scr is:

    login.i

This script starts a `telnet` session, connects to the target system, and logs in on the system using the username and password that were supplied with the `setup` procedure (see section 7.2). A script that is supposed to log in on any UNIX system has to avoid many a pitfall. Some systems ask for a terminal type, other systems require an additional carriage return before presenting the login prompt, and yet other systems put you in the C-shell by default. The `login.i` script is supposed to be fool-proof, but chances are workstation it is not. If a benchmark session gets stuck in its login phase, re-run the benchmark in monitor mode (using the -m option, see chapter 1) and correct any omissions in `login.i`. If this phase succeeds, it is almost certain that the script is connected to the target system and is running the Bourne shell.

The last step in the initial phase is:

    setup.i

This include file sets up the right environment for this session. To avoid flattered response times, this script issues the important command `stty -echo`.

In the second phase, the main script executes a series of include-files. The include files are executed and measured in sequential order. Each include file measures a separate aspect of interactive performance. In constructing the benchmark scenario, we have tried to incorporate as many flavours of interactive usage as possible. We have come up with the following categories/include files:

	`simple.i`	– Simple UNIX commands and common utilities.
	`filters.i`	– Common UNIX filters, such as `awk` and `sed`.
	`pipes.i`	– Pipe throughput.
	`process.i`	– Multi-process forking.
*	`diskio.i`	– Create, copy and sort a big file.
	`ex.i`	– `ex` response time.
	`vi.i`	– `ex` response time.
	`filesys.i`	– File system (including deep directory tree).
	`ftp.i`	– `ftp` small and big file.
	`compile.i`	– `ftp`, compile and run Dhrystone benchmark.

It should be noted that the above selection is somewhat arbitrary. It should also be noted that X-windows- and graphics-oriented activities are not incorporated; the same goes for other activities that deserve the predicate “interactive”, such as accessing files in a distributed file system (NFS).

Space and time are not sufficient to describe each and every script file in detail. The sources of the script files are included in the benchmark file. lt should be noted, that some script files are quintessential to benchmark operation. Critical files are marked with an asterisk (*) in the table above.

The script files make extensive use of shell environment variables. These variables are `HOME`, `REMOTE_HOST`, `REMOTE_USER`, `REMOTE_PASSWORD`, `REMOTE_PROMPT`, `REMOTE_CSHELL_PROMPT`, `WORKSTATION`, `LOGFILE`, and `SESSION_ID`. These variables are automatically initialized by the `setup` and `start` shell scripts. The shell script `sunclient` modifies the variable `$WORKSTATION`. Benchmark results are appended on `$LOGFILE` (response times) and `$LOGFILE.cap` (other data, such as `ftp` transfer rate and Dhrystone performance).

After the include-files have been executed in sequential order, the main script executes the last include-file:

```
wrapup.i
```

which removes remaining files and directories and logs out.

During execution, the script uses files and directories on the target system. To avoid interference with other scripts, the file and directory names contain a unique session identification number. This means that running multiple `bench.scr` scripts on one workstation and one target system is safe. It is pointed out, however, that running multiple sessions on one workstation is likely to influence your measurements.

The elapse time of a single iteration depends on several factors. The major factor is the responsiveness of the target system. Other factors are the amount of error recovery performed by the script and the performance of the network. Observed elapse times range from 2 to 30 minutes in extreme cases.

The disk space occupied by the benchmark results is about 1 Kbyte per single benchmark iteration. This includes the logfile and the capturefile.

`mark.scr`

`mark.scr` is identical to `bench.scr`, except that it does not perform bottleneck-prone network activities to avoid excessive interference with other `bench` or `mark` scripts. The following file is not included in `mark.scr`:

```
ftp.i
```

This include file transfers a big file using ftp; this activity might introduce a communication bottleneck that could afflict the accuracy of the measurements.

By running multiple instances of the `mark.scr` script on multiple workstations, one can attempt to simulate a "representative" interactive workload.

`station.scr`

`station.scr` is a little script that can be used locally to determine some characteristics of the workstation. This script produces a file `logfiles/station.[session-id].log`, which contains relevant workstation data:

- Shell performance.
- Accuracy of workstation's clock.
- Workstation Dhrystone performance.
- Workstation configuration and file system.

8 Run rules and benchmark runs

We recommend to apply the following run rules in order to obtain a complete benchmark with all relevant information:

- The single-user benchmark should be run in first instance "as is". By "as is" we mean that not more than the minimal modifications are made, if any, to execute the programs correctly. If any changes are required, report what is changed.
- If desired, the codes could be optimised. Optimisation is should be only allowed such that these adjustments also can be made by the average user. I.e., by alternate code in a higher programming language or by calls to optimised library routines. Ad hoc assembly code should *not* be allowed. Have any optimisation done reported.

The following runs are expected to be performed:

- The single-user benchmark, "as-is" and optimised (optionally).
- The throughput benchmark. When the codes have been optimised, the optimised versions of the codes could be used. The throughput benchmark is performed according to the `schedule` configuration file (see section 3.5.4).
- The interactive benchmark consists of running:

  1. Running one instance of `bench` on an empty system for 20 iterations.
  2. Running 10 copies of `bench` from multiple workstations on an empty system for 20 iterations. **Note**: The amount of workstations may be less than 10, but make sure that the performance on the individual workstations or the bandwidth between the workstations and the target system is not a limiting factor.
  3. Running 10 copies of `bench` from multiple workstations while the throughput benchmark is running (so, this requires another run of the throughput benchmark).

9 Appendix A — Example session

This appendix presents an example benchmark run that is initiated from a single workstation. In the example,

- the benchmark software is installed and built from scratch;
- the benchmark environment is initialized;
- some characteristics of the workstation are detern-dned;
- three consecutive benchmark iterations are performed;
- the benchmark results are collected analyzed, and displayed, and
- the resuits are saved for future use.

In the text below, the type-ins are again displayed in slanted Courier font such as:

man vi

The workstation's response is displayed in plain Courier, such as:

```
man:  not found
```

We have inserted comment where appropriate. Here we go:
(First, we log in on the front-end workstation under a predefined user name/password:)

```
login:  bench
Password:  geheim
IRIX Release 6.4 IP27 hst3652
Copyright 1987-1997 Silicon Graphics, Inc.  All Rights Reserved.
Last login:  Sat Jun 5 21:56:56 MDT; 1999 by UNKNOWN@aad.phys.uu.nl
$
```

(Subsequently, we remove any existing files and directories:)

```
$ m -rf *; rm -rf .  *
rm:  cannot remove '.'  or '..'
rm:  cannot remove '.'  or '..'
$
```

(Now we `ftp` the `uuencoded` `tar` file from the target system to the workstation. Then, we `uudecode` and unpack the `tar` file and log out from the workstation:)

```
$ ftp 131.211.36.52
Connected to 131.211.36.52.
220 hst3652.fys.ruu.nl FTP server ready.
Name (hst3652:steen):  bench
230 User bench logged in.
ftp> get bench.tar.uue
200 PORT command successful.
150 ASCII data connection for bench.tar.uue (131.211.36.52,3203)
(327346 bytes).
226 ASCII Transfer complete.
local:  bench.tar.uue remote:  bench.tar.uue
32629 bytes received in 0.42 seconds (780 Kbytes/s)
ftp> quit 221 Goodbye.
$ ls
bench.tar.uue
$ uudecode bench.tar.uue
$ ls
bench.tar bench.tar.uue
```

```
$ rm bench.tar.uue
$ xvf bench.tar

 x bin/install, 1150 bytes, 3 tape blocks
 x profile, 267 bytes, 1 tape blocks
 x AAAREADME, 27707 bytes, 55 tape blocks
 x MakF-,file, 256 bytes, 1 tape blocks
 x src/Makefile, 1104 bytes, 3 tape blocks
 x Frc/machine.h, 353 bytes, 1 tape blocks
 x src/script.h, 1619 bytes, 4 tape blocks
 yz src/execute.c, 9258 bytes, 19 tape blocks
 x src/parse.c, 19006 bytes, 38 tape blocks
 x src/proc.c, 4830 bytes, 10 tape blocks
 x src/ptn.c, 5691 bytes, 12 tape blocks
 x src/ptn_spare.c, 4447 bytes, 9 tape blocks
 x src/re-fail.c, 335 bytes, 1 tape blocks
 x src/regex.c, 23468 bytes, 46 tape blocks
 x src/sizes.c, 392 bytes, 1 tape blocks
 x src/tjeg.c, 1186 bytes, 3 tape blocks
 x shellscripts/Makefile, 1376 bytes, 3 tape blocks
 x shellscripts/analyse.sh, 796 bytes, 2 tape blocks
 x shellscripts/graph.sh, 840 bytes, 2 tape blocks
 x shellscripts/run.sh, 3154 bytes, 7 tape blocks
 x shellscripts/save.sh, 2022 bytes, 4 tape blocks
 x shellscripts/setup.sh, 1175 bytes, 3 tape blocks
 x shellscripts/start.sh, 4667 bytes, 10 tape blocks
 x shellscripts/stop.sh, 499 bytes, 1 tape blocks
 x shellscripts/install.sh, 1150 bytes, 3 tape blocks
 x shellscripts/collect.sh, 733 bytes, 2 tape blocks
 x shellscripts/sunclient.sh, 416 bytes, 1 tape blocks
 x utilities/stats.c, 2519 bytes, 5 tape blocks
 x utilities/Makefile, 446 bytes, 1 tape blocks
 x utilities/echobase.c, 526 bytes, 2 tape blocks
 x utilities/bars.c, 2533 bytes, 5 tape blocks
 x utilities/clocktick.c, 1349 bytes, 3 tape blocks
 x sessions/AAAREADME, 1639 bytes, 4 tape blocks
 x sessions/bench.scr, 1374 bytes, 3 tape blocks
 x sessions/dhrystone.c, 19997 bytes, 40 tape blocks
 x sessions/include/simple.i, 2611 bytes, 6 tape blocks
 x sessions/include/process.i, 396 bytes, 1 tape blocks
 x sessions/include/login.i, 1820 bytes, 4 tape blocks
 x sessions/include/vi.i, 744 bytes, 2 tape blocks
 x sessions/include/ftp.i, 761 bytes, 2 tape blocks
 x sessions/include/pipes.i, 674 bytes, 2  tape blocks
 x sessions/include/compile.i, 1221 bytes, 3 tape blocks
 x sessions/include/diskio.i, 1414 bytes,  3 tape blocks
 x sessions/include/filesys.i, 1478 bytes, 3 tape blocks
 x sessions/include/rlogin.i, 1776 bytes,  4 tape blocks
 x sessions/include/filters.i, 716 bytes,  2 tape blocks
 x sessions/include/setup.i, 700 bytes, 2  tape blocks
 x sessions/include/re, 328 bytes, 1 tape  blocks
 x sessions/include/makebig.c, 61 bytes, 1 tape blocks
 x sessions/include/wrapup.i, 910 bytes, 2 tape blocks
 x sessions/include/ping.i, 224 bytes, 1 tape blocks
```

```
 x sessions/include/ex.i, 743 bytes, 2 tape blocks
 x sessions/include/e, 95 bytes, 1 tape blocks
 x sessions/mark.scr, 1375 bytes, 3 tape blocks
 x sessions/station.scr, 1180 bytes, 3 tape blocks
 x sessions/allscripts, 20931 bytes, 41 tape blocks
 x logfiles/dumnLy.log, 388 bytes, 1 tape blocks
```

```
$ ls


AAAREADME     bench.tar      logfiles      shellscripts utilities
makefile      bin            sessions      src
```

```
$ rm bench.tar
$ ^D
```

(We login again to activate the new profile and start the installation procedure:)

```
login:  bench
Password:  geheim
IRIX Release 6.4 IP27 hst3652
Copyright 1987-1997 Silicon Graphics, Inc.  All Rights Reserved.
Last login:  Sat Jun 5 22:35:50 MDT MDT; 1999 by UNKNOWN@aad.phys.uu.nl
$ TERM=xterm
$ export TERM
$ install


 ... Checking environment ...
 ... Building benchmark suite ...
 ... Building C programs
 ... Building utilities
 ... Building stats
 ... Building bars
 ... Building echobase
 ... Building clocktick
 ... Building shell scripts ...
 ... Building start shell script ...
 ... Building setup shell script ...
 ... Building run shell script ...
 ... Building stop shell script ...
 ... Building analyse shell script ...
... Building graph shell script
... Building save shell script
... Building collect shell script ...
... Building sunclient shell script ...
... Build completed ...
    To setup the environment, type: setup
    To start the benchmark, type:
      start [script [logfile [nsessions ...
      - Scripts are any of: bench mark station
      - Logfile is the file to write log data on in directory
logfiles
      - Nsessions is the number of parallel sessions.
    To stop a benchmark, type: stop [script]
    To collect logfiles for further processing, type: collect
```

```
[logfile]
 -> To analyse the results, type: analyse [logfilel or: graph
[logfile]
$
$
```

(Now, we set up a configuration to conform to the actual benchmark environment:)

```
$ setup
Please give the internet address of the target computer:
131.211.37.30
User name on 192.16.188.125 to run benchmarks under:
janus
Password for user janus:
geheim
What is the prompt on 131.211.37.30 in the Bourne shell:
$
What is the prompt on 131.211.37.30 in the C shell:
%
$
```

(In order to determine some characteristics of the front-end workstation we run the station script)

```
$ start station station-1-1
$
990612 223005 Starting run 1 of script station id=28688
$ stop station


Measuring workstations  reaction for shell-prompts
Measuring workstations  reaction for shell-prompts with pattern~
matching
Measuring workstations  info
Measuring workstations  clock ticks
Measuring workstations  dhrystone rate

990612 223201 Run of script: station id=28688 ended ...
```

(Start a series of benchmark runs as follows (script runs as background process:))

```
$
$ start bench bench-se 1


990612 223331 Starting run 1 of script bench id=28738
Logging in on 131.211.37.30 as janus
Logged in on 131.211.37.30 as janus
Setting up environment
- logging response time on file test.log
- stty -echo
Starting interactive benchmark
Results are appended on file locrfiles/bench-se.28738.log
- simple UN*X commands
- awk, sed and other filters
- forking background processes
- compile makebig.c
- create, copy and sort a big file called bigfile
- ex response time
```

```
- vi response time
- mkdir, cd, rm
- mkdir, cd, rm (100)
- FTP-ing a big file
Dhrystone benchmark
- ftp dhrystone.c
- compile dhrystone.c
- run dhrystone
Logged out from 131.211.37.30.
990612 223522 Starting run 2 of script bench id=28738
Logging in on 131.211.37.30 as janus
Logged in on 131.211.37.30 as janus
```

(More runs follow. During the fourth run, we decide it has been enough, and stop the script.)

```
$
$ stop bench
$
990612 224137 script bench id=28738 ended ...
```

(The script has completed. Now, we have to collect our results (even if we did not start multiple sessions!))

```
$
$ collect bench-se
$
Files collected into bench-se.log and bench-se.log.cap
$
ls logfiles
bench-se.log bench-se.log.cap history.giga030-1
```

(We are ready to analyze and diplay our results:)

```
$ analyse bench-se


    4    359       0 awk
    4    470      60 ec
    4   3164     278 cc dhrystone
    4     19       0 cd
    4    564     243 cp bigfile
    4     39       0 date
    4   8450      83 dhrystone
    4     19       0 echo
    4    184      10 echo&
    4   8260     415 ex
    4     44      25 ex browse
    4   4005     164 ex substitutie
    4   13390    158 ftp bigfile
    4    590      11 long filter pipe
    4    540      17 ls -1
    4   2404     517 makebig
    4    224      10 mesg
    4    159       0 mkdir
    4   8725     184 mkdir-100
    4    364      10 ps
    4     19       0 pwd
    4     79       0 rm
```

```
--More--[Press space to continue, 'q' to quit.]  q
```

(More pages of similar information follow. Note that each line shows the number of samples, the average, and the standard deviation of a command)

```
$
$ graph bench-se


 Command: awk
 22:40:06   360
 22:42:44   359
 22:45:23   359
 22:48:02   359
 Command:  cc
 22:40:08   380
 22:42:47   500
 22:45:25   499
 22:48:04   499
 Command: cc dhrystone
 22:41:29 3579
 **************************************************->
 22:44:08 3000
 **************************************************->
 22:46:47 3059
 **************************************************->
 22:49:25 3019
 **************************************************->
 Command:  cd
 22:40:54      19 *
 22:43:33      19 *
 22:46:12      19 *
 22:48:51      19 *
 Command:   cp  bigfile
 22:40:11     520 *************************

--More--[Press space to continue, 'q' to quit.]  q
```

(Convinced that the measurement results are OK, we now save the results to the target system:)

```
$ ls logfiles
bench-se.log bench-se.log.cap history.giga030-1
$
$ save everything


Tar-ing and uuencoding logfile directory to everything.tar.uue
Ftp-ing everything-tar.uue to home directory of janus on 131.211.37.30
...
Done ...

$
$
```

References

[1] A.B. Downey, D.G. Feitelson, *The Elusive Goal of Workload Characterization*, ACM Sigmetrics, **26**, No.4, May 1999, 14–29.

[2] A.J. van der Steen, *The benchmark of the EuroBen Group*, Parallel Computing, **17**, (1991) 1211–1221.