Installation guide for the Sparse Benchmark, version 0.9.7

Victor Eijkhout

17 Nov 2000

1 Introduction

This guide explains the executables, shell scripts, and general structure of the Sparse Iterative Benchmark. This is a set of Fortran77 codes to test the performance of a machine on typical iterative linear system solver operation. The philosophy behind the benchmark is explained in another document.

The basic commands available are:

Configuration of source and shell scripts:
`configure`

Installation of the executables:
`make install`

which creates the executables
`bench_gen bench_sym crs_gen crs_sym reg_gen reg_sym`

Validation of the installation:
`Validate`

Test of performance:
`Test`

Analysis of the results:
`Analyze`

Report of the results to `sparsebench@cs.utk.edu`:
`Report`

Both the `Test` and `Validate` commands start by a call to `make install`.

The structure of the benchmark supports easy exploration of multiple architectures and of multiple variants of the code. Code variants (see section 8) are aimed at exploring different behaviour on mathematically equivalent, but differently implemented, version of the same code.

2 Machine and platform

With one copy of the benchmark package you can test more than one architecture, and more than one variants of the code. These two variables are controlled by using

```
make MACH=<mach> PLAT=<plat> OPT=<opt> install
```
or
```
Install -m <mach> -p <plat> -o <opt>
```
Leaving these variables out leads to defaults
```
MACH=default_machine
PLAT=default_platform
OPT=reference
```
being taken.

The machine name implies nothing more than an identifier for your machine: the platform option indicates what kind of machine you have; see section 3. The optimisation name has to be the name of a subdirectory of SRC; see section 8 on code variants below.

The reason for having both a machine and platform name is that you may want to test two machines of the same architecture, but different clock speeds or cache sizes.

Object files and executables will now be made in the directory
```
SRC/<opt>/<mach>
```
and subsequently linked into the main directory. After an initial make, a subsequent make for that machine is simply a link of the executables.

3 Installation

For compile flags and other platform-specific changes to the makefile, you can

- see if there is a file `Make.<plat>` for your platform, or
- make a copy of `Make.default_platform`,

and edit that file. In most cases this edit is not needed for an initial install, but it will most likely be when you are tuning the performnce.

A shell script detects the architecture, and omitting the platform parameter for a platform for which a `Make.<plat>` file is present for the detected platform, leads to this file being used. Currently, platform-specific instructions are supplied for `default_platform ALPHA HPPA HP300 RS6K SGI5 SGI64`.

The detected platform name can be obtained by `` echo `Scripts/arch` ``; it is a good idea to use this as the extension for any `Make.<plat>` files you create yourself.

Otherwise, use the platform name you choose as the value of the `-p` option to `Install`, or, the `PLAT` option to `make install`, or the `-p` option to one of the other shell scripts (see section 4). Performance will depend on the value of `F_OPT_FLAGS` you set; the `C_OPT_FLAGS` flag only affects the timer and the quicksort routine that is used in the matrix generating part, that is, it does not affect the benchmark performance.

Since this is a F77 code, all allocation is static. To allocate more space, edit the rsize and isize parameters in `main.F` and `main_symm.F`.

The source files all have extension ".F", meaning "fortran77 with preprocessor directives". Some systems think that ".F" means "fortran90". Some of the `Make.<plat>` files contain an attempt to disabuse them of that notion.

The makefile used for the actual installation is `SRC/<opt>/Makefile`. If you inspect the makefile, you will see that the link lines start with an invocation of `$PURIFY`. If you have purify on your system and you run into a bug you can track it down by

```
make clean ; make PURIFY=purify install
```

4 The 'Test', 'Analyze', 'Validate', 'Report' scripts

The benchmark package comes with four shell scripts.

Validate Run some tests and compare the results against results in the file `reference_results`.
Test Run some tests, aimed at getting optimum performance. See section 3 about setting compiler options. See section **??** about generating the test data.
Analyze Post-process the results of the 'Test' script to give benchmark results.
Report Bundle up the results in a file, and send it to `sparsebench@cs.utk.edu`.

4.1 Specifying machine name and type

If you want to test more than one machine, do

```
Validate -m <machine name> -p <platform>
```

and

```
Test -m <machine name> -p <platform>
```

which will use the file `Make.<platform>` and leave the results in `Out/<machine name>`, and subsequently

```
Analyze -m <machine name>
```

The 'Analyze' command writes to the screen, so just capture the results however you want. If you leave out the `-m` option, `default_machine` is taken as the architecture name; if you leave out the `-p` option, `default_platform` is taken as the platform name.

The 'Test' and 'Validate' scripts also take a code variant argument (see section 8), for instance

```
Test -p mta -o wave_ilu
```

This does not change the location of where test results are stored.

The 'Test' and 'Validate' scripts start out by calling

```
make PLAT=<platform> MACH=<machine> OPT=<opt> install
```

where the platform, machine, and optimisation have been specified by `-p`, `-m` and `-o` options or are taken as the default.

4.2 Test data and analysis

The 'Test' script runs every problem a number of times; by default 3 times. You can alter this number by `Test -r <n>`. In each run, the script tests whether a certain problem has already been run, and will only overwrite results if the new run gives higher performance.

Since the full set of test matrices will take a few hundred megabyte of disc space, matrices are deleted after use. If you want to test several machines, want to save the time it takes to generate them repeatedly, and have a few hundred megabyte to spare, use Test -d which saves the matrices to disc.

The ‘Analyze’ script has an option to limit the results that are analyzed. For instance

```
Analyze -c gmres
```

displays only timings for files in Out/<platform> that have gmres in the name.

You can get a plot of the analysis data by specifying

```
Analyze -d
```

Since this will currently draw 12 plots in one figure, it is a good idea to combine this with the -c option to limit what you are plotting. The -d option relies on ‘gnuplot’ (version 3.5 is too old, 3.7 works, I have not tested 3.6) and ‘ghostview’ being in your path.

The ‘Analyze’ script uses a combination of awk and perl scripts, and a small Fortran program Scripts/lsq.f, which gets made the first time you call ‘Analyze’. You can specify a -p option to make sure the correct Make.<platform> file is used in the compilation.

Sometimes, the ‘Analyze’ script will report a flop rate of zero, especially for the vector operations. This is a consequence of the timer being used, which has on most platforms a resolution of 1/100 second. Often, operations will take less than this. In that case, time values of zero will be reported as a zero flop rate.

4.3 Benchmark reporting

The ‘Report’ script bundles up all files in Out/<machine name> into a file ‘Results.<machine>’, and sends this by email to sparsebench@cs.utk.edu. There are a few options to modify the behaviour of this script:

-m ¡name¿ Report for machine name; leaving this out causes ‘default˙machine’ to be used.

-p ¡plat¿ Include information that the platform is plat; leaving this out causes the result of Scripts/arch to be used.

-c ¡component¿ Report only results that match component; for instance, Report -c gmres causes only GMRES results to be reported.

-n Do not use email. By default, the bundled reports in ‘Results.<machine>’ are sent using

```
mail sparsebench@cs.utk.edu < \verb+Results.<machine>+
```

Using the ‘-n’ option omits this step, and it is up to you to get the file to us somehow.

The ‘Report’ script asks you a few questions, such as a description of your machine, and whether the code was compiled straight out of the box, or with modifications applied. This should all be fairly obvious.

5 Test data

There are programs crs_gen and reg_gen to generate unsymmetric, and crs_sym and reg_sym to generate symmetric matrices, of crs and diagonal storage respecitively, and write them to file. The bench_* programs will detect these dumps and read them.

Both the Test and Validate scripts call these auxiliaries, so they generate some large temporary files with names `crsmat*` and `regmat*`. If you want to test more than once, or more than one architecture, it is a good idea to leave these test matrices around; normally they are deleted immediately. If you have plenty of disk space, use

```
Test -d
```

to have the matrices saved after use.

Generating the test matrices may take quite some time if you are running larger problems. There is an option

```
-s "size1 size2 ... "
```

to indicate which sizes are to be tested or validated. By default, validation uses sizes `10 20`, while testing is done on the sizes `12 14 16 18 20 24 28 32 36 38`. You will see an error message if you try to validate sizes for which no reference data is in the file `reference_results`.

6 Output

The benchmark code runs for 10 iterations of an iterative method, printing the residual error in each iteration. Especially if you specify a large matrix size, you may not see the error go down by much in each iteration. Not to worry. This is because the iterative method would need hundreds of iterations to converge, and we are only interested in benchmarking the per iteration performance.

7 Documentation

In addition to the README file, this installation guide, you can also read:

- `bench.ps` about the philosophy of the benchmark, and lists of results, and
- `generate.ps` about the random matrix generator.

8 Code variants

A small number of lightly optimised code variants are provided. These can be constructed by

```
make OPT=name install
```

Object files and executables are left in directories

```
SRC/<OPT>/<ARCH>
```

see section 2.

The currently available variants are:

`naive_ilu` naive coding of regular ilu solve, with bound checking tests inside the inner loop.
`wave_ilu` wavefront coding of regular ilu solve.
`long_vector` contiguous storage of diagonals in regular storage.
`bulkgmres` implementation of gmres using QR after building Krylov space.

`bulkgmres_lapack` as above, but using lapack routines where possible.
`classical_gs` implementation of gmres using classical Gram-Schmidt.
`cprod` matrix-vector product in C.
`reference` the reference code.

9 Executables

The installation leaves the executables

`bench_gen bench_sym crs_gen crs_sym reg_gen reg_sym`

in the current directory.

`bench_gen` code for general linear systems, choice between BiCG and GMRES methods

`bench_sym` code for symmetric systems, only half the matrix is stored; only CG.

The same names are used for the code variants, so you have to do a `make OPT=<opt> install` in between tests of different variants.