Iterative Solver Benchmark

Jack Dongarra, Victor Eijkhout, Henk van der Vorst

2001/01/14

1 Introduction

The traditional performance measurement for computers on scientific application has been the Linpack benchmark [2], which evaluates the efficiency with which a machine can solve a dense system of equations. Since this operation allows for considerable reuse of data, it is possible to show performance figures a sizeable percentage of peak performance, even for machines with a severe unbalance between memory and processor speed.

In practice, sparse linear systems are equally important, and for these the question of data reuse is more complicated. Sparse systems can be solved by direct or iterative methods, and especially for iterative methods one can say that there is little or no reuse of data. Thus, such operations will have a performance bound by the slower of the processor and the memory, in practice: the memory.

We aim to measure the performance of a representative sample of iterative techniques on any given machine; we are not interested in comparing, say, one preconditioner on one machine against another preconditioner on another machine. Therefore we have strict conformance tests on the results of the benchmark, explained below.

An earlier report on the performance of supercomputers on sparse equation solvers can be found in [3].

2 Motivation

The sparse benchmark offers a small number of iterative methods, preconditioners and storage schemes. Rather than implementing all known methods, we have chosen those that have a performance representative of larger classes. More detailed discussion will follow in section 4.

We want to stress from the outset that we did not aim to present the most sophisticated method. Rather, by considering combinations of the representative elements used in the benchmark a user should be able to get a good notion of the expected performance of methods not included. Consistent with this philosophy, we terminate each benchmark run after a fixed number of iterations, since we are not interested in convergence speed: we solely measure the flop rate per iteration.

As storage schemes we offer diagonal storage, and compressed row storage. Both of these formats represent typical matrices for three-dimensional finite element or finite difference methods. The diagonal storage, using seven diagonals, is the natural mode for problems on a regular ('brick') domain; the compressed row storage[1] is the natural storage scheme for irregular domains. Thus these choices are representative for most single-variable physical problems.

The iterative methods provided are CG and GMRES. The plain Conjugate Gradients method is representative of all fixed-storage methods, including sophisticated methods for nonsymmetric problems such as BiCGstab; the GMRES method represents the class of methods that have a storage demand that grows with the number of iterations.

Each iterative method can be run unpreconditioned – which is computationally equivalent to using a Jacobi preconditioner – or with an ILU preconditioner. For the diagonal storage scheme a block Jacobi method is also provided; this gives a good indication of domain decomposition methods. If these methods are used with inexact subdomain solves, then the ILU preconditioner gives the expected performance for these.

3 Structure of the benchmark

We have implemented a benchmark that constructs a test matrix and preconditioner, and solves a linear system with them. Separate flop counters and timers are kept for the work expended in vector operations, matrix vector products, preconditioner solves, and other operations involved in the iterative method. The flop counts and flops rates in each of these categories, as well as the overall flops rates, are reported at the end of each run.

The benchmark comprises several storage formats, iterative methods, and preconditioners. Together these form a representative sample of the techniques in typical sparse matrix applications. We describe these elements in more detail in section 4.

We offer a reference code, which is meant to represent a portable implementation of the various methods, without any machine-specific optimisations. In addition to this we supply a number of variants of the code that should perform better on certain machines, and most likely worse on some others; see section 6.

3.1 Conformance of the benchmark

Since we leave open the possibility that a local implementer make fargoing changes to the benchmark code (see section 3.2), every submission of benchmark results has to be accompanied by proof that the local implementation conforms to the reference code. For conformance testing we supply a script that matches the error norm in the final iteration of a run (we stop the iteration after ten steps) to the value of that error stored on file.

This is a static test, and in practice it only allows changes to the reference code that are not numerically significant. In particular, it precludes an implementer from replacing the preconditioner by a different one. We justify this from our standpoint

1. Use of compressed column storage should give roughly the same performance.

that the benchmark is not a test of the best possible preconditioner or iterative method, but rather of methods representative for a wider class with respect to computer performance.

Since the benchmark includes ILU preconditioners, this static conformance test would a priori seem to be biased against parallel implementations of the benchmark. This point is further elaborated in section 5.

3.2 Benchmark reporting

An implementer of the benchmark can report performance results on various levels, each next level encompassing all of the earlier options.

1. Using only compiler flags in the compilation of the reference code.
2. Using compiler directives in the source of the reference code.
3. Rewriting the reference code in such a way that any differences are solely in a different order of scheduling the operations.
4. Rewriting the reference code by replacing some algorithm by a mathematically equivalent formulation of the algorithm (that is: in exact arithmetic the (intermediate) results should be the same).

The last two levels may or will in general influence the numerical results, so results from codes thus rewritten should be accompanied by proof that the specific realisation of the benchmark reproduces the reference results within a certain tolerance.

Each run of the benchmark code ends with a report on how many floating point operations were performed in the various operations. Implementers should use these numbers to do reporting (rather than using hardware flop counters, for instance), but they are free to substitute their own timers.

The benchmark comes with shell scripts that run a number of tests, and report both best performance and asymptotic performance for the whole code and elements of it. Asymptotic performance is determined by making a least-squares fit $y = a+bx^{-1}$ through the data points, where y is the observed megaflop rate and x is the dataset size. The asymptotic performance is then the value of a.

This assumption on the performance behaviour accomodates both cache-processors, for which we expect $b > 0$ as the dataset size overflows the cache, and vector processors, for which we expect $b < 0$ as performance goes up with increasing vector length.

For cache-based processors we may expect a plateau behaviour if the cache is large; we discard the front of this plateau when calculating the asymptotic performance.

4 Elements of the benchmark code

The user of the benchmark has the following choices in determining the problem to run.

4.1 Storage formats

The matrix can be in the following formats:

- Diagonal storage for a seven-diagonal matrix corresponding to finite differences in three dimensions;
- Compressed row storage of a matrix where the sparsity structure is randomly generated; each row has between 2 and 20 nonzeros, each themselves randomly generated, and the bandwidth is $\leq n^{2/3}$ which again corresponds to a problem in three space dimensions.

For both formats a symmetric variant is given, where only half the matrix is stored.

The diagonal storage is very regular, giving code that has a structure of loop nests of depth three. Vector computers should perform very efficiently on this storage scheme. In general, all index calculation of offsets can be done statically.

Matrix-vector operations on compressed row storage may have a different performance in the transpose case from the regular case. Such an operation in the regular case is based on inner products; in the transpose case it uses vector updates (axpy operations). Since these two operations have different load/store characteristics, they may yield different flops rates. In the symmetric case, where we store only half the matrix, such operations use in fact the regular algorithm for half the matrix, and the transpose algorithm for the other half. Thus, the performance of, for instance, the matrix-vector product, will be different in GMRES from in the Conjugate Gradient method.

The CRS format gives algorithms that consist of an outer loop over the matrix rows, with an inner loop that involves indirect addressing. Thus, we expect a lower performance, especially on machines where the indirect addressing involves an access to memory.

4.2 Iterative methods

The following iterative methods have been implemented[2] (for more details on the methods mentioned, see the Templates book [1]):

- Conjugate Gradients method; this is the archetypical Krylov space method for symmetric systems. We have included this, rather than MINRES or SYMLQ, for its ease of coding, and for the fact that its performance behaviour is representative of the more complicated methods. The results for CG are also more-or-less representative for transpose-free methods for nonsymmetric systems, such as BiCGstab, which also have a storage demand constant in the number of iterations.
- Generalized Minimum Residual method, GMRES. This popular method has been included because its performance behaviour is different from CG: storage and computational complexity are an increasing function of the iteration count. For that reason GMRES is most often used in cycles of m steps. For low values of m, the computational performance for GMRES will not be much different than for CG. For larger values, say $m > 5$, the j inner products in the j-th iteration may influence the performance. We have included GMRES(20) in our benchmark.

2. We have not included methods such as BiCG in the benchmark, which use a product with the transpose matrix A^t. In many cases forming this product is impractical, and for this reason such methods are less used than transpose-free methods such as BiCGstab. BiCG has the same performance behaviour as CG, except for the difference between the regular and the transpose matrix-vector product. For diagonal matrix storage there is no difference; for compressed storage it is the difference between a dot product and a vector update, both indirectly addressed.

The Conjugate and BiConjugate gradient methods (see figure 1) involve, outside

Let A, M, x, b be given;
compute $r_1 = Ax - b$;
for $i = 1 \ldots 10$
    solve preconditioner: $z = M^{-1}r$
    inner product $\rho_i = r^t z$
    if $i > 1$, update $p \leftarrow z + p(\rho_i/\rho_{i-1})$
    matrix vector product: $q = Ap$
    inner product $\pi = p^t q$
    update $x \leftarrow x - p\pi$
        $r \leftarrow r - q\pi$

Figure 1: Conjugate Gradient algorithm

the matrix-vector product and preconditioner application, only simple vector operations. Thus, their performance can be characterised as Blas1-like. The GMRES method (see figure 2), on the other hand, uses orthogonalisation of each new gen-

Let A, M, x, b be given;
for $i = 1 \ldots 10$
    matrix and preconditioner apply: $z = AM^{-1}r$
    orthogonalize z against all earlier v_j, $j < i$
    normalize $v_i \leftarrow z/\|z\|$.
    update QR factorisation of size $i + 1 \times i$ Hessenberg matrix
Update $x \leftarrow x - \sum_i v_i c_i$

Figure 2: One restart cycle of the Generalized Minimum Residual method

erated Krylov vector against all previous, so a certain amount of cache reuse should be possible. See also section 6 for a rewritten version that uses Blas3 kernels.

4.3 Preconditioners

The following preconditioners are available[3]:

- No preconditioner;
- Point ILU; for the diagonal storage a true ILU-D is implemented, in the CRS case we use SSOR, which has the same algorithmic structure as ILU;
- Line ILU for the diagonal storage scheme only; this makes a factorisation of the line blocks.
- Block Jacobi for the diagonal storage scheme only; this is parallel on the level of the plane blocks. The block Jacobi preconditioner gives a performance representative of domain decomposition methods, including Schwarz methods.

The point ILU method is typical of commonly used preconditioners. It has largely the structure of the matrix-vector product, but on parallel machines its sequential nature inhibits efficient execution.

3. We have not included the commonly used Jacobi preconditioner, since this is mathematically equivalent to scaling the matrix to unit diagonal, a strategy that has the exact same performance as using no preconditioner.

The line ILU method uses a Level 2 BLAS kernel, namely the solution of a banded system. It is also a candidate for algorithm replacement, substituting a Neumann expansion for the system solution with the line blocks.

5 Parallel realisation

Large parts of the benchmark code are conceptually parallel. Thus we encourage the submission of results on parallel machines. However, the actual implementation of the methods in the reference code is sequential. In particular, the benchmark includes ILU preconditioners using the natural ordering of the variables.

It has long been realised that ILU factorisations can only be implemented efficiently on a parallel architecture if the variables are renumbered from the natural ordering to, for instance, a multi-colour or nested dissection ordering.

Because of our strict conformance test (see section 3.1), the implementer is not immediately at liberty to replace the preconditioner by an ILU based on a different ordering. Instead, we facilitate the parallel execution of the benchmark by providing several orderings of the test matrices, namely:

- Reverse Cuthill-McKee ordering.
- Multi-colour ordering; here we do not supply the numbering with the minimal number of colours, but rather a colouring based on [4].
- Nested dissection ordering; this is an ordering based on edge-cutting, rather than finding a separator set of nodes.

The implementer then has the freedom to improve parallel efficiency by optimising the implementation for a particular ordering.

Again, the implementer should heed the distinction of section 3.2 between execution by using only compiler flags or directives in the code, and explicit rewrites of the code to force the parallel distribution.

6 Code variants

We supply a few variants of the reference code that incorporate transformations that are unlikely or impossible to be done by a compiler. These transformations target specific architecture types, possibly giving a higher performance than the reference code, while still conforming to it; see section 3.1.

Naive coding of regular ILU Putting the tests for boundary conditions in the inner loop is bad coding practice, except for dataflow machines, where it exposes the structure of the loop.

Wavefront ordering of regular ILU We supply a variant of the code where the triple loop nest has been rearranged explicitly to a sequential outer loop and two fully parallel inner loops. This may benefit dataflow and vector machines.

Long vectors At the cost of a few superfluous operations on zeros, the vector length in the diagonal-storage matrix-vector product can be increased from $O(n)$ to $O(n^3)$. This should benefit vector computers.

Different GMRES orthogonalisation algorithms There are at least two reformulations of the orthogonalisation part of the GMRES method. They can enable use of Level 3 BLAS operations and, in parallel context, combine inner product operations. However, these code transformations no longer preserve the semantics under computer – rather than exact – arithmetic.

7 Results

The following tables contain preliminary results for the machines listed in table 1. In table 2 we report the top speed reported regardless the iterative method, preconditioner, and problem size. This speed is typically reported on a fairly small problem, where presumably the whole data set fits in cache.

We compute an 'asymptotic speed' by a least-squares fit as described in section 3.2. We report this asymptotic speed for the following components:

- The matrix vector product. We report this in regular storage, and in compressed row storage separately for the symmetric case (cg) and the nonsymmetric case since these may have different performance characteristics; see section 4.1.
- The ILU solve. We also report this likewise in three variants.
- The Block Jacobi solve.
- Vector operations. These are the parts of the algorithm that are independent of storage formats. We report the efficiency of vector operations for the CG method; in case of GMRES a higher efficiency can be attained by using Level 2 BLAS and Level 3 BLAS routines. We have not tested this.

Processor	Manufacturer / type	Clock speed / #procs	Compiler / options
Alpha EV56	Compaq	433 MHz	f77 -O5
Alpha EV67	Compaq	500 MHz	f77 -O5
Athlon	AMD	1 GHz	g77 -O
LX164	DCG	533 MHz	g77 -O5
MIPS R10000	SGI Indigo	195 MHz	f77 -O
MIPS R12000	SGI Octane	dual 270 MHz	f77 -O
PIII	Dell	550 MHz	g77 -O
P4	Dell	1.5GHz	g77 -O
Power3	IBM	dual 200 MHz	xlf -O
PowerPC G4	Apple	450 MHz	g77 -O
UltraSparcII	Sun	quad 296 MHz	f77 -O

Table 1: List of machines used

We see from the results that the performance of these sparse operations, in contrast to the dense operations in for instance the Linpack benchmark, is almost completely determined by the quality of the memory subsystem.

8 Obtaining and running the benchmark

The benchmark code can be obtained from `http://www.netlib.org/benchmark/sparsebench`. The package contains Fortran code, and shell scripts for installation and postprocessing. Results can be reported automatically to `sparsebench@cs.utk.edu`, which address can also be used for questions and comments.

Machine	Mflop/s
EV67	705
Power3	550
P4	425
EV56	331
UltraSparcII	259
LX164	233
MIPS R12000	228
PowerPC G4	163
PIII	163
Athlon	163
MIPS R10000	151

Table 2: Highest attained performance

Machine	Mflop/s
EV67	86
P4	73
Power3	68
MIPS R12000	49
Athlon	45
UltraSparcII	40
LX164	34
EV56	34
PIII	24
PowerPC G4	21
MIPS R10000	18

Table 3: Asymptotic performance of Diagonal storage Matrix-vector product

References

[1] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia PA, 1994. `http://www.netlib.org/templates/templates.ps`.

[2] Jack Dongarra. Performance of various computers using standard linear equations software. http://www.netlib.org/benchmark/performance.ps.

[3] Jack Dongarra and Henk van der Vorst. Performance of various computers using sparse linear equations software in a fortran environment. *Supercomputer*, 1992.

[4] M.T. Jones and P.E. Plassmann. A parallel graph coloring heuristic. *SIAM J. Sci. Stat. Comput.*, 14, 1993.

Machine	Mflop/s
EV67	76
Power3	58
P4	46
MIPS R12000	39
Athlon	34
EV56	28
LX164	24
PIII	22
UltraSparcII	18
MIPS R10000	16
PowerPC G4	9

Table 4: Asymptotic performance of Symmetrically stored CRS Matrix-vector produc

Machine	Mflop/s
P4	97
EV67	96
MIPS R12000	53
Athlon	39
LX164	28
EV56	28
UltraSparcII	27
PIII	27
PowerPC G4	19
MIPS R10000	19
Power3	14

Table 5: Asymptotic performance of CRS Matrix-vector product

Machine	Mflop/s
EV67	86
MIPS R12000	41
Athlon	34
LX164	29
EV56	28
P4	24
Power3	21
PIII	19
PowerPC G4	13
MIPS R10000	9
UltraSparcII	7

Table 6: Asymptotic performance of Diagonal storage ILU solve

Machine	Mflop/s
P4	70
EV67	69
Power3	57
Athlon	31
MIPS R12000	26
LX164	26
EV56	26
PIII	21
PowerPC G4	19
UltraSparcII	17
MIPS R10000	10

Table 7: Asymptotic performance of Symmetrically stored CRS ILU solve

Machine	Mflop/s
P4	86
EV67	48
Power3	45
Athlon	21
MIPS R12000	18
LX164	16
EV56	15
PIII	13
UltraSparcII	8
MIPS R10000	6
PowerPC G4	5

Table 8: Asymptotic performance of CRS ILU solve

Machine	Mflop/s
EV67	201
Power3	113
MIPS R12000	113
P4	79
EV56	65
MIPS R10000	58
UltraSparcII	47
PIII	44
LX164	41
PowerPC G4	26
Athlon	10

Table 9: Asymptotic performance of Vector operations in CG

Machine	Mflop/s
EV67	106
MIPS R12000	65
Power3	53
P4	51
LX164	33
EV56	32
Athlon	30
PIII	18
UltraSparcII	16
PowerPC G4	12
MIPS R10000	12

Table 10: Asymptotic performance of Diagonal storage Block Jacobi solve