An Iterative Solver Benchmark

Lapack working note 152

Jack Dongarra, Victor Eijkhout, Henk van der Vorst*

revision 2001/06/01

Abstract

We present a benchmark of iterative solvers for sparse matrices. The benchmark contains several common methods and data structures, chosen to be representative of the performance of a large class of commonly used methods. We give results on some high performance processors that show that performance is largely determined by memory bandwidth.

1 Introduction

In scientific computing, several benchmarks exist that give a user some idea of the to-be-expected performance given a code and a specific computer. One widely accepted performance measurement is the Linpack benchmark [4], 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 that are a sizeable percentage of peak performance, even for machines with a severe unbalance between memory and processor speed.

However, sparse linear systems are at least as important in scientific computing, 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. In fact, the range of possible preconditioners is so large, and their performance so much dependent on the specific problem, that we do not even compare one preconditioner to another on the same machine. Instead, we identify kernels that will have a representative performance, and test those; the performance of a whole code is then a composite of the performances of the various sparse kernels.

An earlier report on the performance of supercomputers on sparse equation solvers can be found in [5].

* This research is sponsored in part by Subcontract #R71700J-29200099 with William Marsh Rice University and Subcontract #B503962 with The Regents of the University of California.

2 Motivation

Iterative methods are hard to benchmark. The ultimate measure one is interested in is 'time expended to reach a certain accuracy'. This number is a function of at least the following variables:

1. The iterative method: for each type of linear system (symmetric / hermitian / general, definite / almost definite / indefinite, et cetera) there are several candidate methods with differing mathematical and computational characteristics. For instance, GMRES [9] has as advantage over BiCGstab [10] that it minimizes residuals, and is able to use Level 3 BLAS; it has as disadvantage that work per iteration is higher and memory demands larger.
2. The preconditioner: some systems are well-conditioned enough that a simple Jacobi preconditioner (or equivalently, diagonal scaling before the iteration process) will be good enough, others require complicated preconditioners to tame the spectrum.
3. Data structures used: the same matrix can be stored in a number of formats, some of which may have computational advantages. For instance, problems with several physical variables per node can be expressed in such a way that dense matrix operations can be used.
4. The implementation of computational kernels: different implementations of the same operation can have different performance characteristics.
5. Architectural details of the machine used, in particular the processor speed and the memory bandwidth. Also, depending on the operation, the one or the other may be of more or less importance.
6. Computer Arithmetic: some problems maybe solvable to a desired accuracy in single precision, others may require double precision. Even at a given precision, two different implementations of arithmetic may give different results, so that the same program on two different machines will not behave the same.
7. The problem to be solved. The behaviour of all of the above points is a function of the problem to be solved. The number of iterations (that is, behaviour of the iterative method and preconditioner) is a function of the spectrum of the matrix. The flop rate per iteration (determined by implementation of kernels, and processor characteristics) is a function of the sparsity structure of the matrix.
8. Problem size. Even ignoring such obvious factors as performance degradation because of page swapping, the size of a problem's data set can influence performance strongly. On vector machines, increasing the problem size will most likely increase the vector length and hence performance. Cache-based machines will reach optimum performance for a problem that fits in cache. Some architectures (notable the Alpha chip) have a large enough cache that realistic problems can actually be solved in-cache. In general, however, increasing the problem size will always overflow the cache, and iterative methods have in general little opportunity for cache reuse. Hence we expect decreasing performance to some 'asymptotic' limit determined more by bandwidth than by processor speed.

Given this plethora of variables, it should be clear that the desired metric is unfortunately not a benchmarkable one.

This situation contrasts starkly with that of dense, direct, solvers. There, the only problem parameter determining performance is the size of the system. Thus, it is

truly possible to benchmark a machine.

In order to provide a comprehensive iterative benchmark, a package would have to cover the range of choices of items 1–3. Taking into account possible criticism of the quality of implementation used (item 4), we see that a comprehensive benchmark is almost equivalent to a library of all high-quality implementations of all existing methods. Clearly, this is not a practical proposition. Even if we were to provide such a large collection of well-implemented methods, we would have to provide a set of standard problems (this is necessary in order to make comparisons between machines), and this may very well overlook a particular kind of problem the user is interested in, and which has its own set of performance characteristics.

Instead of even attempting the above-sketched perfect benchmark, we have opted for providing a small set of iterative methods, preconditioners and storage schemes, on the criterion that they have a performance representative of larger classes. We then envision the following scenario for the use of our benchmark: user solve a few representative test problem with their preferred solver, thus gaining a notion of convergence speed in terms of numbers of iterations. They can then consult our benchmark to gauge the per-iteration performance of their solver, which combined with their prototype run on the test problem gives an accurate impression of the expected running time.

We want to stress from the outset that we did not aim to present the most sophisticated methods. 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. In effect, our benchmark only indicates performance per occurrence of a computational kernel. Users can then estimate their codes' performance by combining these numbers, applied to their specific implementation, with the numbers of iterations needed for their problem.

Consistent with this philosophy, we terminate each benchmark run after a fixed number of iterations. The number of iterations to the desired accuracy is mostly a function of the problem, and only to a minor extent of the computer architecture. It is almost independent of the computational kernels that are the subject of this benchmark, since these can only influence the convergence through different round-off behaviour. For the most part we will assume that different implementations of the same kernel have roughly the same round-off characteristics, so that our per-iteration measurement is a sufficient predictor of the overall efficiency.

To account for the effects of caching in our benchmark, we run a series of problems of increasing size for each choice of method, preconditioner, and storage scheme. In this sequence we measure both the maximum attained performance, often a clear function of cache size, and we estimate an 'asymptotic' performance as the problem scales up; see sections 3.2 and 7.

We conclude this section by giving a brief description of the sparse kernels. More detailed discussion will follow in section 4.

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

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

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 such methods are used with inexact subdomain solves, 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. The reference code is written in Fortran, using double precision arithmetic throughout; implementers are not allowed to increase speed by switching to single precision. In addition to the reference code we supply a number of variants 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), we need to ascertain that the optimised code still conforms to the original, that is, that the implementer has only optimised the *implementation* of the algorithms for a specific machine, and has not *replaced* one algorithm by another. Unfortunately, by the nature of iterative methods, this is a hard problem.

Iterative methods such as Conjugate Gradients are forward unstable; they do not have a self-correction mechanism the way stationary iterative methods have. This means that comparing the computed quantities of one method against another, or, if it were possible, comparing it against an exact arithmetic method, would be essentially meaningless. Likewise, any bounds from using interval arithmetic would be so large as to be useless. The reason that conjugacy-based iterative methods are

usefull despite this unstable behaviour, is that a more careful analysis shows that accumulated roundoff does not cause divergence, but will only delay convergence by a modest number of iterations (see [6, 7]).

Thus we are in the predicament of having to enforce the numerical conformance of a method that is intrinsically unstable in the naive sense of the term. Our practical test, therefore, checks how much an optimised code differs from the original after computing one iteration, that is, before any large divergence from exact arithmetic starts to show up.

There are several ways this test can be implemented specifically, and none of them are totally satisfying. Absent an implementation in exact arithmetic, we basically have the choice of comparing optimised code against a reference code on the same machine, and code on some reference machine. For the first option, we would basically trust that arithmetic is implemented in a reasonable manner on machines available today, so that the reference code, without any optimisations that alter the straightforward execution of instructions, is indeed a reasonable reference point. The second option raises the question what machine would be trustworthy enough to function as a supplier of reference results. We can evade that question by observing that, under the above assumption that all machines on the market today have a 'reasonable' implementation of arithmetic, the difference between two concrete implementations of the benchmark is bounded by the sum of the differences between either and an exact arithmetic implementation. Our measurement of difference against a result on file – generated by us, the producers of the benchmark – would then give a small overestimate of the true error.

This latter strategy, comparison against results on file, is the one we have chosen. (We are in good company with this decision: the NAS parallel benchmarks [1] use the same verification scheme, and in fact with a far stricter test than ours.) In practice, we test whether the deviation remains under 100 times the machine precision. The number '100' itself is somewhat arbitrary; it reflects the limited number of nonzeros in each matrix row, but it does not reflect the worst-case error bound of $O(N)$ that comes in through the inner product calculations in the iterative method. A test after one iteration could conceivably be performed against an interval arithmetic implementation, but we have not done so.

We emphasize that this is a static test, designed to allow only 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 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 (for more details on the methods mentioned, see the Templates book [2]):

- 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.
- BiConjugate Gradients. In many ways this method has the same performance characteristics as CG, but it differs in that it additionally uses a product with the transpose matrix A^t. In many cases forming this product is impractical, and for this reason BiCG and such methods as QMR are less used than transpose-free methods such as BiCGstab. We have included it nevertheless, since the performance of this kernel can not be derived from others. For diagonal matrix storage there is actually no difference between the regular and transpose matrix-vector product; for compressed storage it is the difference between a dot product and a vector update, both indirectly addressed.
- Generalized Minimum Residual method, GMRES. This popular method has been included because its performance behaviour is very 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.

The Conjugate and BiConjugate gradient methods (see figure 1) involve, outside the matrix-vector product and preconditioner application, only simple vector operations. Thus, their performance can be characterised as similar to that of Level 1 BLAS. The GMRES method (see figure 2), on the other hand, uses orthogonalisation of each new generated 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 Level 3 BLAS kernels.

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

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

4.3 Preconditioners

The following preconditioners are available[2]:

- 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.

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.

2. 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.

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 [8].
- 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.

All results are double precision, that is, 64-bit arithmetic.

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.
- 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.

The results published here are only preliminary; a fuller database – both with more machines and of more aspects of the methods tested, as well as code variants – will be maintained on netlib.

We report both 'highest performance on any problem' and 'asymptotic performance', but the former number is of limited value. On superscalar machines like the ones tested here – and unlike on vector machines to which we had no ready access – it is of necessity reached for a small, hence unrealistic problem. In particular on machines with a relatively small cache the number reported is further unreliable since small problems easily drop under the timer resolution. We will fix this in later version by offering PAPI [3] support and adaptively raising the maximum number of iterations[3] to match the timer resolution.

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. The highest performance in obtained in the vector operations, closely followed by the diagonal-storage matrix-vector product, as these are the only operations that do not involve indirection due to general sparse storage.

Beyond such simple observations, however, performance becomes harder to analyse, a fact that is already familiar from dense benchmark suites. For instance, on most machines, the regular CRS matrix-vector product predictably performs slighly better than the transpose product, which involves more memory references. However, the product with a symmetrically stored matrix, which in implementation is a regular product with the upper triangle and a transpose product with the lower triangle, has a lower performance than either. The CRS ILU solve, in turn, which has a very similar structure to the matrix-vector product, has these rankings exactly reverse for a few machines. A full analysis of such results would go far beyond the scope of this paper.

3. Note however, that this will only increase the reliability of the performance of the whole problem: the times for the individual components will be almost vanishingly small.

Processor	Manufacturer / type	Clock rate (MHz)	peak Mflops rate	Compiler / options
Alpha EV67	Compaq	500	1000	f77 -O5
Athlon	AMD	1000	2000	g77 -O3 -malign-double -funroll-loops
MIPS R12000	SGI Octane	270	1080	f77 -Ofast
PIII	Dell	550	550	g77 -O3 -malign-double -funroll-loops
P4	Dell	1500	1500	g77 -O3 -malign-double -funroll-loops
Power3	IBM	200	800	xlf -O4
UltraSparcII	Sun	296	296	f77 -fast

Table 1: List of machines used

Machine	Mflop/s
Compaq Alpha EV67	932
IBM Power3	807
Intel Pentium III	386
Intel Pentium 4	259
Athlon	235
MIPS R12000	219
UltraSparcII	146

Table 2: Highest attained performance

Machine	Mflop/s
IBM Power3	183
Compaq Alpha EV67	129
Athlon	60
UltraSparcII	55
MIPS R12000	53
Intel Pentium III	46
Intel Pentium 4	41

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

Machine	Mflop/s
IBM Power3	117
Compaq Alpha EV67	117
Intel Pentium 4	88
Athlon	46
MIPS R12000	39
Intel Pentium III	34
UltraSparcII	23

Table 4: Asymptotic performance of CRS Matrix-vector product

Machine	Mflop/s
IBM Power3	92
Compaq Alpha EV67	76
Intel Pentium 4	56
Athlon	29
Intel Pentium III	24
MIPS R12000	22
UltraSparcII	16

Table 5: Asymptotic performance of Symmetrically stored CRS Matrix-vector product

Machine	Mflop/s
IBM Power3	108
Compaq Alpha EV67	99
Intel Pentium 4	67
MIPS R12000	45
Athlon	44
Intel Pentium III	26
UltraSparcII	19

Table 6: Asymptotic performance of Transpose CRS Matrix-vector product

Machine	Mflop/s
Compaq Alpha EV67	100
IBM Power3	97
Intel Pentium 4	49
MIPS R12000	45
Intel Pentium III	32
Athlon	32
UltraSparcII	24

Table 7: Asymptotic performance of Diagonal storage ILU solve

Machine	Mflop/s
Intel Pentium 4	57
Compaq Alpha EV67	57
IBM Power3	40
Athlon	27
Intel Pentium III	21
MIPS R12000	17
UltraSparcII	10

Table 8: Asymptotic performance of CRS ILU solve

Machine	Mflop/s
Compaq Alpha EV67	81
IBM Power3	64
Intel Pentium 4	44
Athlon	34
Intel Pentium III	28
MIPS R12000	25
UltraSparcII	19

Table 9: Asymptotic performance of Symmetrically stored CRS ILU solve

Machine	Mflop/s
IBM Power3	64
Intel Pentium 4	55
Compaq Alpha EV67	50
MIPS R12000	30
Athlon	26
Intel Pentium III	19
UltraSparcII	12

Table 10: Asymptotic performance of Transpose CRS ILU solve

Machine	Mflop/s
Compaq Alpha EV67	262
IBM Power3	258
MIPS R12000	142
Intel Pentium 4	110
Intel Pentium III	86
Athlon	68
UltraSparcII	59

Table 11: Asymptotic performance of Vector operations in CG

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 post-processing. Results can be reported automatically to `sparsebench@cs.utk.edu`, which address can also be used for questions and comments.

References

[1] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The NAS parallel benchmarks. *Intl. Journal of Supercomputer Applications*, 5:63–73, 1991.

[2] 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`.

[3] S. Browne, J Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*.

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

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

[6] A. Greenbaum. *Iterative Methods for Solving Linear Systems*. SIAM, Philadelphia, 1997.

[7] A. Greenbaum and Z. Strakos. Predicting the behavior of finite precision Lanczos and Conjugate Gradient computations. pages 121–137, 1992.

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

[9] Yousef Saad and Martin H. Schultz. GMRes: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, 1986.

[10] Henk van der Vorst. Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13:631–644, 1992.