International Journal of High Performance Computing Applications
http://hpc.sagepub.com/

The Problem With the Linpack Benchmark 1.0 Matrix Generator

Jack J. Dongarra and Julien Langou
International Journal of High Performance Computing Applications 2009 23: 5
DOI: 10.1177/1094342008098683

The online version of this article can be found at:
http://hpc.sagepub.com/content/23/1/5

Published by:

http://www.sagepublications.com

Additional services and information for *International Journal of High Performance Computing Applications* can be found at:

Email Alerts: http://hpc.sagepub.com/cgi/alerts

Subscriptions: http://hpc.sagepub.com/subscriptions

Reprints: http://www.sagepub.com/journalsReprints.nav

Permissions: http://www.sagepub.com/journalsPermissions.nav

Citations: http://hpc.sagepub.com/content/23/1/5.refs.html

Downloaded from hpc.sagepub.com at UNIV OF TENNESSEE on June 21, 2011

THE PROBLEM WITH THE LINPACK BENCHMARK 1.0 MATRIX GENERATOR

Jack J. Dongarra[1, 2, 3,]
Julien Langou[4]

Abstract

We characterize the matrix sizes for which the Linpack Benchmark 1.0 matrix generator constructs a matrix with identical columns.

Key words: Linpack benchmark, pseudo-random number generator, random matrix, HPL, TOP500

The International Journal of High Performance Computing Applications, Volume 23, No. 1, Spring 2009, pp. 5–13
DOI: 10.1177/1094342008098683
© 2009 SAGE Publications Los Angeles, London, New Delhi and Singapore
Figures 1, 4–15 appear in color online: http://hpc.sagepub.com

1 Introduction

Since 1993, twice a year, a list of the sites operating the 500 most powerful computer systems has been released by the TOP500 project (http://www.top500.org/). A single number is used to rank computer systems based on the results obtained on the *High Performance Linpack Benchmark* (HPL Benchmark).

The HPL Benchmark consists of solving a dense linear system in double precision, 64-bit floating point arithmetic, using Gaussian elimination with partial pivoting. The ground rules for running the benchmark state that the supplied matrix generator, which uses a pseudo-random number generator, must be used in running the HPL benchmark. The supplied matrix generator can be found in *High Performance Linpack 1.0* (HPL-1.0; http://www.netlib.org/benchmark/hpl/) which is an implementation of the HPL Benchmark. In a HPL benchmark program, the correctness of the computed solution is established and the performance is reported in floating point operations per sec (flops/sec). It is this number that is used to rank computer systems across the world in the TOP500 list. For more on the history and motivation for the HPL Benchmark, see Dongarra, Luszczek, and Petitet (2003).

In May 2007, a large high performance computer manufacturer ran a 20-hour-long HPL Benchmark. The run failed with the output result:

```
|| A x - b ||_oo / (eps * ||A||_1 * N)
    = 9.224e+94...... FAILED
```

It turned out that the manufacturer chose n to be $n = 2{,}220{,}032 = 2^{13} \cdot 271$. This was a bad choice. In this case, the HPL Benchmark 1.0 matrix generator produced a matrix A with identical columns. Therefore the matrix used in the test was singular and one of the checks of correctness determined that there was a problem with the solution and the results should be considered questionable. The reason for the suspicious results was neither a hardware failure nor a software failure but a predictable numerical issue.

Nick Higham pointed out that this numerical issue had already been detected in 1989 for the LINPACK-D benchmark implementation, a predecessor of HPL, and had been reported to the community by David Hough (1989).

[1] DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE, UNIVERSITY OF TENNESSEE

[2] OAK RIDGE NATIONAL LABORATORY

[3] UNIVERSITY OF MANCHESTER

[4] DEPARTMENT OF MATHEMATICAL AND STATISTICAL SCIENCES, THE UNIVERSITY OF COLORADO DENVER (JULIEN.LANGOU@UCDENVER.EDU)

Downloaded from hpc.sagepub.com at UNIV OF TENNESSEE on June 21, 2011

Another report has been made to the HPL developers in 2004 by David Bauer with $n = 131{,}072$. In this manuscript, we explain why and when the Linpack Benchmark 1.0 matrix generator generates matrices with identical columns. We define $\mathcal{S}$ as the set of all integers such that the Linpack Benchmark 1.0 matrix generator produces a matrix with at least two identical columns. We characterize and give a simple algorithm to determine if a given n is in $\mathcal{S}$.

Definition 1. *We define $\mathcal{S}$ as the set of all integers such that the Linpack Benchmark 1.0 matrix generator produces a matrix with at least two identical columns. For $i > 2$, we define $\mathcal{S}_i$ as the set of all integers such that the Linpack Benchmark 1.0 matrix generator produces a matrix with at least one column repeated i times.*

In Table 1, for illustration, we give the 40 smallest integers in $\mathcal{S}$ along with the largest i for which the associated matrix size is in $\mathcal{S}_i$.

Some remarks are in order.

Remark 1.1. If $i > j > 2$ then $\mathcal{S}_i \subset \mathcal{S}_j \subset \mathcal{S}$.

Remark 1.2. If n is in $\mathcal{S}$, then the matrix generated by the Linpack Benchmark 1.0 matrix generator has at least two identical columns, therefore this matrix is necessarily singular. If n is not in $\mathcal{S}$, the coefficient matrix has no identical columns; however, we do not claim that the matrix is nonsingular. Not being in $\mathcal{S}$ is not a sufficient condition for being nonsingular.

Remark 1.3. In practice, we would like the coefficient matrix to be well-conditioned (since we want to numerically solve a linear system of equations associated with them). This is a stronger condition than being nonsingular. Edelman (1988) proves that for real n-by-n matrices with elements from a standard normal distribution, the expected value of the log of the 2-norm condition number is asymptotic to $\log n$ as $n \to \infty$ (roughly $\log n + 1.537$). The Linpack Benchmark 1.0 matrix generator uses a uniform distribution on the interval $[-0.5, 0.5]$, for which the expected value of the log of the 2-norm condition number is also asymptotic to $\log n$ as $n \to \infty$ (roughly $4\log n + 1$), see Cuesta-Albertos and Wschebor (2003). Random matrices are expected to be well-conditioned; however, pseudo-random number generators are only an attempt to create randomness and we will see that, in some particular cases, the generated matrices have repeated columns and are therefore singular (that is to say infinitely ill-conditioned).

Remark 1.4. HPL-1.0 checks whether a zero-pivot occurs during the factorization and reports it to the user. As a result of rounding errors, even if the initial matrix has two identical columns, exact-zero pivots hardly ever occur in practice. Consequently, it is difficult for benchmarkers to distinguish between numerical failures and hardware/software failures. This issue is further investigated in Section 5.

Remark 1.5. In Remark 1.3, we stated that we would like the coefficient matrix to be well-conditioned. Curiously enough, we will see in Section 5 that the HPL benchmark can successfully return when run on a matrix with several identical columns. This is because the combined effect of finite precision arithmetic (that transforms a singular matrix into an ill-conditioned matrix) and the use of a test for correctness that is independent of the condition number of the coefficient matrix.

Table 1
The 40 matrix sizes smaller than 500,000 for which the Linpack Benchmark 1.0 matrix generator will produce a matrix with identical columns. The number in parenthesis indicates the maximum number of times each column is repeated. For example, the entry "491,520 (8)" indicates that, for the matrix size 491,520, there exists one column that is repeated eight times while there exists no column that is repeated nine times.

65,536 (2)	98,304 (2)	131,072 (8)	147,456 (2)	163,840 (3)
180,224 (2)	196,608 (6)	212,992 (2)	229,376 (4)	245,760 (2)
262,144 (32)	270,336 (2)	278,528 (3)	286,720 (2)	294,912 (5)
303,104 (2)	311,296 (3)	319,488 (2)	327,680 (10)	335,872 (2)
344,064 (3)	352,256 (2)	360,448 (6)	368,640 (2)	376,832 (3)
385,024 (2)	393,216 (24)	401,408 (2)	409,600 (4)	417,792 (2)
425,984 (7)	434,176 (2)	442,368 (4)	450,560 (2)	458,752 (14)
466,944 (2)	475,136 (4)	483,328 (2)	491,520 (8)	499,712 (2)

Downloaded from hpc.sagepub.com at UNIV OF TENNESSEE on June 21, 2011

2 How the Linpack Benchmark Matrix Generator Constructs a Pseudo-Random Matrix

The pseudo-random coefficient matrix A from the HPL Benchmark 1.0 matrix generator is generated by the HPL subroutine `HPL_pdmatgen.c`. In this subroutine, the pseudo-random number generator uses a linear congruential algorithm (see for example Knuth 1997, Section 3.2)

$$X(n + 1) = (a * X(n) + c) \bmod m,$$

with $m = 2^{31}$, $a = 1103515245$, $c = 1235$. These choices of m, a, and c are fairly standard and we find them, for example, in the standard POSIX.1-2001 or in the GNU libc library for the `rand()` function. The maximum period of a sequence generated by a linear congruential algorithm is at most m, and in our case, with HPL-1.0's parameters a and c, we indeed obtain the maximal period 2^{31}. (Proof: either by direct check or using the Full-Period Theorem, see Knuth 1997, Section 3.2). This provides us with a periodic sequence s such that $s(i + 2^{31}) = s(i), for any i \in \mathbb{N}$. HPL-1.0 fills its matrices with pseudo-random numbers by columns using this sequence s starting with $A(1, 1) = s(1)$, $A(2, 1) = s(2)$, $A(3, 1) = s(3)$, and so on.

Definition 2. *We define a Linpack Benchmark 1.0 matrix generator, a matrix generator such that*

$$A(i, j) = s((j - 1) * n + i), \quad 1 \le i, j \le n \tag{1}$$

and s is such that

$$s(i + 2^{31}) = s(i), \quad \text{for any } i \in \mathbb{N} \text{ and } \quad s(i) \neq s(j), \quad \text{for any } 1 \le i, j \le 2^{31} \tag{2}$$

Some remarks:

Remark 2.1. The assumption $s(i) \neq s(j)$, for any $1 \le i, j \le 2^{31}$ is true in the case of the Linpack Benchmark 1.0 matrix generator. It can be relaxed to admit more sequences s for which some elements can be identical. However, this assumption makes the sufficiency proof of the theorem in Section 4 easier and clearer.

Remark 2.2. It is important to note that the matrix generated by the Linpack Benchmark 1.0 matrix generator solely depends on the dimension n. The Linpack Benchmark 1.0 matrix generator requires benchmarkers to use the same matrix for any block size, for any number of processors or for any grid size.

Remark 2.3. Moreover, since the Linpack Benchmark 1.0 matrix generator possesses its own implementation of the pseudo-random number generator, the computed pseudo-random numbers in the sequence s depend weakly on the computer systems. Consequently the pivot pattern of the Gaussian elimination is preserved from one computer system to another, from one year to another.

Remark 2.4. Finally, the linear congruential algorithm for the sequence s enables the matrix generator for a scalable implementation of the construction of the matrix: each process can generate their local part of the global matrix without communicating or generating the global matrix. This property is not usual among pseudo-random number generators.

Remark 2.5. To give a sense of the magnitude of the size n of matrices, the matrix size for the #1 entry in the TOP500 list of June 2008 was 2,236,927 which is between 2^{21} and 2^{22}. The smallest matrix size in the TOP500 list of June 2008 was 273,919 which is between 2^{18} and 2^{19}.

Remark 2.6. The pseudo-random number generator has been changed five times in the history of the Linpack Benchmark. We recall here some historical facts.

1980 – LINPACKD-1.0 – The initial LINPACKD benchmark uses a matrix generator based on the (Fortran) code below:

```
   subroutine matgen(n,a,lda)
   real a(lda,*)
   init = 1325
   do 10 j = 1,n
      do 20 i = 1,n
         init = mod(3125*init,65536)
         a(i,j) = (init - 32768.0)/16384.0
20    continue
10 continue
   end
```

The period of this pseudo-random number generator is: $2^{14} = 16{,}384$.

1989 – Numerical failure report – David Hough (1989) observed a numerical failure with the LINPACKD-1.0 benchmark for a matrix size $n = 512$ and submitted his problem as an open question to the community through NA-Digest.

1989 – LINPACKD-2.0 – Two weeks after David Hough's post, Robert Schreiber (1989) posted in NA-Digest an explanation of the problem, he gave credit to Nick Higham and himself for the explanation. The problem #27.4 in Nick Higham's (2002) book *Accuracy and Stability of*

Downloaded from hpc.sagepub.com at UNIV OF TENNESSEE on June 21, 2011

Numerical Algorithms is inspired by this story. Higham and Schreiber also provide a patch to improve the pseudo-random number generator. Replacing line 6 of the previous code

```
init = mod(3125*init,65536)
```

by

```
init = mod(3125*init-1,65536)
```

increases the period from $2^{14} = 16{,}384$ to $2^{16} = 65{,}536$. We call this version LINPACKD-2.0.

1992 – LINPACKD-3.0 – The pseudo-random number generator of LINPACKD is updated for good in 1992 by using the DLARUV LAPACK routine based on Fishman's (1990) multiplicative congruential method with modulus 2^{48} and multiplier 33952834046453.

2000 – HPL-1.0 – First release of HPL (09/09/2000). The pseudo-random number generator uses a linear congruential algorithm (see for example Knuth 1997, Section 3.2)

$$X(n + 1) = (a * X(n) + c) \bmod m,$$

with $m = 2^{31}$, $a = 1103515245$, $c = 1235$. The period of this pseudo-random number generator is 2^{31}.

2004 – Numerical failure report – Gregory Bauer observed a numerical failure with HPL and $n = 2^{17} = 131{,}072$. History repeats itself. The HPL developers recommended to HPL users willing to test matrices of size larger than 2^{15} to not use power two.

2007 – Numerical failure report – A large manufacturer observed a numerical failure with HPL and $n = 2{,}220{,}032$. History repeats itself again. Note that $2{,}200{,}032 = 2^{13} \cdot 271$, and is not a power of two.

2008 – HPL-2.0 – This present manuscript explains the problem in the Linpack Benchmark 1.0 matrix generator. As of September 10th 2008, Piotr Luszczek has incorporated a new pseudo-random number generator in HPL-2.0. This pseudo-random number generator uses a linear congruential algorithm with $a = 6364136223846793005$, $c = 11$ and $m = 2^{64}$. The period of this pseudo-random number generator is 2^{64}.

3 Understanding $\mathcal{S}$

Consider a large dense matrix of order $3 \cdot 10^6$ generated by the process described in Definition 2. The number of entries in this matrix is $9 \cdot 10^{12}$ which is above the pseudo-random number generator period ($2^{31} \approx 2.14 \cdot 10^9$). However, despite this fact, it is fairly likely for the constructed matrix to have distinct columns and even to be well-conditioned.

On the other hand, we can easily generate a "small" matrix with identical columns. Take $n = 2^{16}$, we have for any $i = 1, \ldots, n$:

$$\begin{aligned} A(i, 2^{15} + 1) &= s(i + n * (j - 1)) \\ &= s(i + 2^{15} * n) = s(i + 2^{15} * 2^{16}) \\ &= s(i + 2^{31}) = s(i) = A(i, 1), \end{aligned}$$

therefore the column 1 and the column $2^{15} + 1$ are exactly the same. The column 2 and the column $2^{15} + 2$ are exactly the same, etc. We can actually prove that $2^{16} = 65{,}536$ is the smallest matrix order for which a multiple of a column can happen.

Another example of $n \in \mathcal{S}$ is $n = 2^{31} = 2{,}147{,}483{,}648$ for which all columns of the generated matrix are the same. Our goal in this section is to build more n in $\mathcal{S}$ to have a better knowledge of this set.

If n is a multiple of $2^0 = 1$ and $n > 2^{31}$ then $n \in \mathcal{S}$. (Note that the statement "any n is a multiple of $2^0 = 1$ and $n > 2^{31}$" means $n > 2^{31}$.) The reasoning is as follows. There are 2^{31} indexes from 1 to 2^{31}. Since there are at least $2^{31} + 1$ elements in the first row of A (assumption $n > 2^{31}$), then, necessarily, at least one index (say k) is repeated twice in the first row of A. This is the pigeonhole principle. Therefore we have proved the existence of two columns i and j such that they both start with the kth term of the sequence. If two columns start with the index of the sequence, they are the same (since we take the element of the column sequentially in the sequence). The three smallest numbers of this type are

$$n = 2^0 * (2^{31} + 1) = 2{,}147{,}483{,}649 \in \mathcal{S}$$

$$n = 2^0 * (2^{31} + 2) = 2{,}147{,}483{,}650 \in \mathcal{S}$$

$$n = 2^0 * (2^{31} + 3) = 2{,}147{,}483{,}651 \in \mathcal{S}.$$

If n is a multiple of $2^1 = 2$ and $n > 2^{30}$ then $n \in \mathcal{S}$. If n is even ($n = 2q$), then the first row of A accesses the numbers of the sequence s using only odd indexes. There are 2^{30} odd indexes between 1 and 2^{31}. Since there are at least $2^{30} + 1$ elements in the first row of A (assumption $n > 2^{30}$), then, necessarily, at least one index is repeated twice in the first row of A. This is the pigeonhole principle. The three smallest numbers of this type are:

Downloaded from hpc.sagepub.com at UNIV OF TENNESSEE on June 21, 2011

$$n = 2^1 * (2^{29} + 1) = 1{,}073{,}741{,}826 \in \mathcal{S}$$

$$n = 2^1 * (2^{29} + 2) = 1{,}073{,}741{,}828 \in \mathcal{S}$$

$$n = 2^1 * (2^{29} + 3) = 1{,}073{,}741{,}830 \in \mathcal{S}.$$

If n is a multiple of $2^2 = 4$ and $n > 2^{29}$ then $n \in \mathcal{S}$. If n is a multiple of 4 ($n = 4q$), then the first row of A accesses the numbers of the sequence s using only $(4q + 1)$–indexes. There are 2^{29} $(4q + 1)$–indexes between 1 and 2^{31}. Since there are at least $2^{29} + 1$ elements in the first row of A (assumption $n > 2^{29}$), then, necessarily, at least one index is repeated twice in the first row of A. This is the pigeonhole principle. The first three numbers of this type are:

$$n = 2^2 * (2^{27} + 1) = 536{,}870{,}916 \in \mathcal{S}$$

$$n = 2^2 * (2^{27} + 2) = 536{,}870{,}920 \in \mathcal{S}$$

$$n = 2^2 * (2^{27} + 3) = 536{,}870{,}924 \in \mathcal{S}.$$

$$\vdots$$

If n is a multiple of 2^{13} and $n > 2^{18}$ then $n \in \mathcal{S}$. This gives for example:

$$n_{12} = 2^{13} * (2^5 + 1) = 2^{13} * 33 = 270{,}336 \in \mathcal{S}$$

$$n_{13} = 2^{13} * (2^5 + 2) = 2^{13} * 34 = 278{,}528 \in \mathcal{S}$$

$$n_{15} = 2^{13} * (2^5 + 3) = 2^{13} * 35 = 294{,}912 \in \mathcal{S}.$$

These three numbers correspond to entries (3, 2), (3, 3), and (3, 5) in Table 1.

If n is a multiple of 2^{14} and $n > 2^{17}$ then $n \in \mathcal{S}$. This gives for example:

$$n_4 = 2^{14} * (2^3 + 1) = 2^{14} * 9 = 147{,}456 \in \mathcal{S}$$

$$n_5 = 2^{14} * (2^3 + 2) = 2^{14} * 10 = 163{,}840 \in \mathcal{S}$$

$$n_6 = 2^{14} * (2^3 + 3) = 2^{14} * 11 = 180{,}224 \in \mathcal{S}.$$

These three numbers correspond to entries (1, 4), (1, 5), and (2, 1) in Table 1.

If n is a multiple of 2^{15} and $n > 2^{16}$ then $n \in \mathcal{S}$. This gives for example:

$$n_2 = 2^{15} * (2^1 + 1) = 2^{15} * 3 = 98{,}304 \in \mathcal{S}$$

$$n_3 = 2^{15} * (2^1 + 2) = 2^{15} * 4 = 131{,}072 \in \mathcal{S}$$

$$n_5 = 2^{15} * (2^1 + 3) = 2^{15} * 5 = 163{,}840 \in \mathcal{S}.$$

These three numbers correspond to entries (1, 2), (1, 3), and (1, 5) in Table 1.

If n is a multiple of 2^{16} and $n > 2^{15}$ then $n \in \mathcal{S}$.

$$n_1 = 2^{16} * (2^0 + 1) = 2^{16} * 1 = 65{,}536 \in \mathcal{S}$$

$$n_3 = 2^{16} * (2^0 + 2) = 2^{16} * 2 = 131{,}072 \in \mathcal{S}$$

$$n_7 = 2^{16} * (2^0 + 3) = 2^{16} * 3 = 196{,}608 \in \mathcal{S}.$$

These three numbers correspond to entries (1, 1), (1, 3), and (2, 2) in Table 1.

From this section, we understand that any n multiple of 2^k and larger than 2^{31-k} is in $\mathcal{S}$. In the next paragraph, we prove that these are indeed the only integers in $\mathcal{S}$, which provides us with a complete characterization of $\mathcal{S}$.

4 Characterization of $\mathcal{S}$

Theorem. *$n \in \mathcal{S}$ if and only if the matrix of size n generated by the Linpack Benchmark 1.0 matrix generator has at least two identical columns if and only if*

$$n > 2^{31-k} \quad \text{where } n = 2^k \cdot q \text{ with } q \text{ odd.}$$

Proof.

$\Leftarrow$ Let us assume that n is a multiple of 2^k, that is to say

$$n = 2^k \cdot q,\ 1 \le q$$

and let us assume that

$$n > 2^{31-k}.$$

In this case, the first row of A accesses the numbers of the sequence s using only $(2^k \cdot q + 1)$–indexes. There are 2^{31-k} $(2^k \cdot q + 1)$–indexes between 1 and 2^{31}. Since there are at least $2^{31-k} + 1$ elements in the first row of A (assumption $n > 2^{31-k}$), then, necessarily, at least one index is repeated twice in the first row of A. This is the pigeonhole principle. If two columns start with the same index in the sequence, they are the same (since we take the element of the column sequentially in the sequence).

$\Rightarrow$ Assume that there are two identical columns i and j in the matrix generated by the Linpack Benchmark 1.0 matrix generator ($i \ne j$). Without loss of generality, assume $i > j$. The fact that column i is the same as column j means that these columns have identical

Downloaded from hpc.sagepub.com at UNIV OF TENNESSEE on June 21, 2011

entries, in particular, they share the same first entry. We have

$$A(1, i) = A(1, j).$$

From this, equation (1) implies

$$s(1 + (i - 1)n) = s(1 + (j - 1)n).$$

Equation (2) states that all elements in a period of length 2^{31} are different, therefore, since $i \neq j$, we necessarily have

$$1 + (i - 1)n = 1 + (j - 1)n + 2^{31} \cdot p, \quad 1 \leq p.$$

This implies

$$(i - j)n = 2^{31} \cdot p, \quad 1 \leq p.$$

We now use the fact that $n = 2^k \cdot q$ with q odd and get

$$(i - j) \cdot 2^k \cdot q = 2^{31} \cdot p, \quad 1 \leq p, q \text{ is odd}.$$

Since q is odd, this last equality implies that 2^{31} is a divisor of $(i - j) \cdot 2^k$. This writes

$$(i - j) \cdot 2^k = 2^{31} \cdot r, \quad 1 \leq r.$$

From which, we deduce that

$$(i - j) \cdot 2^k \geq 2^{31}.$$

A upper bound for i is n, a lower bound for j is 1; therefore,

$$(n - 1) \cdot 2^k \geq 2^{31}.$$

We conclude that, if a matrix of size n generated by the Linpack Benchmark 1.0 matrix generator has at least two identical columns, this implies

$$n > 2^{31-k} \text{ where } n = 2^k \cdot q \text{ with } q \text{ odd}.$$

□

5 Solving (Exactly) Singular System in Finite Precision Arithmetic with a Small Backward Error

From our analysis, the first matrix size n for which the Linpack Benchmark 1.0 matrix generator will generate a matrix with two identical columns is n = 65,536 (see Table 1). However, HPL-1.0 passes all the tests for correctness on this matrix size. It is the same for n = 98,304 which is our second matrix size in the list (see Table 1). If we look more carefully at the output file for n = 2,220,032, we see that only one out of the three tests for correctness is triggered:

```
||Ax-b||_oo / (eps * ||A||_1 * N)
    = 9.224e+94 ...... FAILED
||Ax-b||_oo / (eps * ||A||_1 * ||x||_1)
    = 0.0044958 ...... PASSED
||Ax-b||_oo / (eps * ||A||_oo * ||x||_oo)
    = 0.0000002 ...... PASSED
```

Despite the fact that the matrix has identical columns, we observe that HPL-1.0 is sometimes able to pass all the tests, sometimes two tests out of three and sometimes none of the three tests. This section will answer how this behavior is possible. First of all, we need to explain how the Linpack Benchmark assesses the correctness of an answer.

5.1 How the Linpack Benchmark Program Checks a Solution

To verify the result after the LU factorization, the benchmark regenerates the input matrix and the right-hand side, then an accuracy check on the residual $Ax - b$ is performed.

The LINPACKD benchmark checks the accuracy of the solution by returning

$$\frac{\|Ax - b\|_\infty}{n\varepsilon\|A\|_M\|x\|_\infty}$$

where $\|A\|_M = \max_{i,j}|a_{ij}|$ and ε is the relative machine precision.

For HPL-1.0, the three following scaled residuals are computed:

$$r_n = \frac{\|Ax - b\|_\infty}{n\varepsilon\|A\|_1},$$

$$r_1 = \frac{\|Ax - b\|_\infty}{\varepsilon\|A\|_1\|x\|_1},$$

$$r_\infty = \frac{\|Ax - b\|_\infty}{n\varepsilon\|A\|_\infty\|x\|_\infty}.$$

A solution is considered numerically correct when all of these quantities are less than a threshold value of 16. The last quantity (r_∞) corresponds to the normwise backward error in the infinite norm allowing perturbations on A only (Higham 2002). The last two quantities (r_∞, r_1) are

Downloaded from hpc.sagepub.com at UNIV OF TENNESSEE on June 21, 2011

Table 2
The three entries in the TOP500 June 2008 list with suspicious *n*.

Rank	Site	Manufacturer	Year	NMax
16	Information Technology Center, The University of Tokyo	Hitachi	2008	1,433,600 (6)
49	The Earth Simulator Center	NEC	2002	1,075,200 (2)
88	Cardiff University – ARCCA	Bull SA	2008	634,880 (2)

independent of the condition number of the coefficient matrix A and should always be less than a threshold value of the order of 1 (no matter how ill-conditioned A is).

For HPL-2.0, the check for correctness is

$$r_4 = \frac{\|Ax - b\|_\infty}{n\varepsilon(\|A\|_\infty\|x\|_\infty + \|b\|_\infty)}. \quad (3)$$

This corresponds to the normwise backward error in the infinite norm allowing perturbations on A and b only (Higham 2002). A solution is considered numerically correct when this quantity is less than a threshold value of 16. Although the error analysis of Gaussian elimination with partial pivoting can be done in such a way that b is not perturbed (in other words r_∞ is the criterion you want to use for Gaussian elimination with partial pivoting), HPL-2.0 switches to r_4, the usual backward error as found in textbooks.

This discussion on the check for correctness explains why HPL-1.0 is able to pass the test for correctness even though the input matrix is exactly singular.

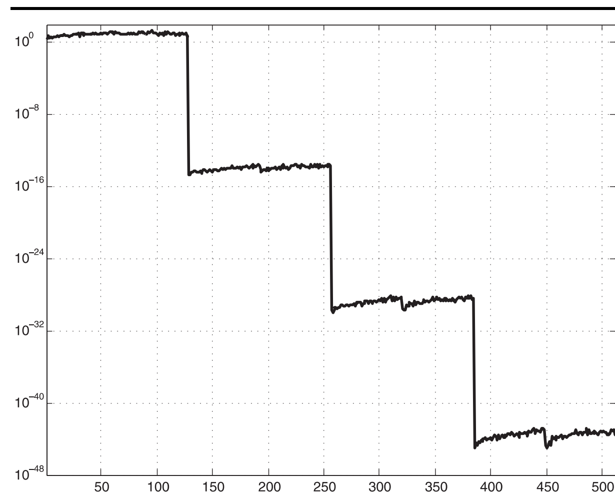

Fig. 1 Magnitude of the pivot (diagonal entries along the matrix *U*) for $n = 512 = 2^9$ and the LINPACK-2.0 matrix generator. The period of the LINPACK-2.0 matrix generator is $n = 65536 = 2^{16}$ so that, for a matrix of size $n = 512$, columns repeat every 128 columns. We observe that pivots are multiplied by $\varepsilon \approx 2.2 \cdot 10^{-16}$ at every repetition.

5.2 Repeating Identical Blocks to the Underflow

Schreiber and Higham (1989) explain what happens when a block is repeated k times in the initial coefficient matrix A. At each repeat, the magnitude of the pivot (diagonal entries of the U matrix) are divided by ε. This is illustrated in Figure 1. This process continues until underflow occurs. Denormalized numbers might help but the process is still the same and ultimately a zero pivot is reached, and the algorithm is stopped. In single precision arithmetic with $\varepsilon_s = 2^{-24}$ and underflow 2^{-126}, five identical blocks will lead to underflow. In double precision arithmetic with $\varepsilon = 2^{-16}$ and underflow 2^{-1022}, one will need 64 identical blocks.

5.3 Anomalies in Matrix Sizes Reported in the June 2008 TOP500 List

Readers of this manuscript may be surprised to find three entries in the TOP 500 data from June 2008 with matrix sizes that lead to matrices with identical columns if the HPL test matrix generator is used. These three entries are given in Table 2. For example, the run for the Earth Simulator from 2002 was done with n = 1,075,200 which corresponds to $2^{11} \cdot 525$, therefore, the column $j = 2^{20}$ = 1,048,576 would have been a repeat of the first under our assumptions. The benchmark run on the Earth Simulator in 2002 was done with an older version of the test harness. This test harness predates the HPL test harness and uses another matrix generator than the one provided by HPL. Today we require the HPL test harness to be used in the benchmark run.

6 How to Fix the Problem

Between 1 and $1 \cdot 10^6$, there are 49 matrix sizes in $\mathcal{S}$ (see Table 1). Between 1 and $3 \cdot 10^6$, there are 1,546 matrix sizes in $\mathcal{S}$. Therefore, for this order of matrix size, there is

Downloaded from hpc.sagepub.com at UNIV OF TENNESSEE on June 21, 2011

a good chance of choosing a matrix size that is not in $\mathcal{S}$. Unfortunately benchmarkers tend to pick multiples of high power of 2 for their matrix sizes which increases the likelihood of picking an $n \in \mathcal{S}$.

1. The obvious recommendation is to choose any n as long as it is odd. In the odd case if $n < 2^{31} \approx 4 \cdot 10^9$, then $n \notin \mathcal{S}$.
2. A check can be added at the beginning of the execution of the Linpack Benchmark matrix generator. The C-code looks as follows:

```
long long int m,n;
int i,k,t,s;
s = 31;
m=n; k=0; while (m%2==0)
    {k++; m=m/2;}
m=1; t=0; while (m<=n) {t++; m=m*2;}
if (t+k>s) i = 1; else i = 0;
```

   n is the matrix size, 2^s is period of the pseudo-random number generator ($s = 31$ in our case) and i is the output flag. If $i = 1$, then $n \in \mathcal{S}$. If $i = 0$, then $n \notin \mathcal{S}$. (The check could also consist of looking over a table.)
3. If $n \in \mathcal{S}$, one can simply pad the matrix with an extra line. This can be easily done in the HPL code `HPL_pdmatgen.c` by changing the variable `jump3` from `M` to `M+1` whenever $n \in \mathcal{S}$.
4. Another possibility is to increase the period of the pseudo-random number generator used. For example, if the pseudo-random number generator had a period of 2^{64} and if $n \le 2^{32}$, then, assuming $(i \neq j \Rightarrow s(i) \neq s(j))$, entries would never repeat.
5. A check for correctness robust to ill-conditioned matrix could be used as discussed in Section 5.

The problem with the Linpack Benchmark 1.0 matrix generator is now corrected in the Linpack Benchmark 2.0 matrix generator. The fix includes both proposition 4 (extend the period of the pseudo-random generator) and proposition 5 (have a test for correctness robust to ill-conditioned matrices).

Acknowledgments

The authors would like to thank Piotr Luszczek and Antoine Petitet for their valuable comments on HPL, Nick Higham for making us aware of David Hough's (1989) *Random Story* and his comments on the backward error analysis of Gaussian elimination with partial pivoting, and finally Asim Yarkhan for one pertinent observation.

Author Biographies

Jack Dongarra received a B.Sc. in mathematics from Chicago State University in 1972 and an M.Sc. in computer science from the Illinois Institute of Technology in 1973. He received his Ph.D. in applied mathematics from the University of New Mexico in 1980. He worked at the Argonne National Laboratory until 1989, becoming a senior scientist. He now holds an appointment as University Distinguished Professor of Computer Science in the Computer Science Department at the University of Tennessee and holds the title of Distinguished Research Staff in the Computer Science and Mathematics Division at Oak Ridge National Laboratory (ORNL), Turning Fellow at Manchester University, and an Adjunct Professor in the Computer Science Department at Rice University. He is the director of the Innovative Computing Laboratory at the University of Tennessee. He is also the director of the Center for Information Technology Research at the University of Tennessee which coordinates and facilitates IT research efforts at the University. He specializes in numerical algorithms in linear algebra, parallel computing, the use of advanced-computer architectures, programming methodology, and tools for parallel computers. His research includes the development, testing and documentation of high quality mathematical software. He has contributed to the design and implementation of the following open source software packages and systems: EISPACK, LINPACK, the BLAS, LAPACK, ScaLAPACK, Netlib, PVM, MPI, NetSolve, Top500, ATLAS, and PAPI. He has published approximately 200 articles, papers, reports and technical memoranda and he is co-author of several books. He was awarded the IEEE Sid Fernbach Award in 2004 for his contributions in the application of high performance computers using innovative approaches and in 2008 he was the recipient of the first IEEE Medal of Excellence in Scalable Computing. He is a Fellow of the AAAS, ACM, and IEEE and a member of the National Academy of Engineering.

Julien Langou received two M.Sc. degrees from the Ecole National Superieure de l'Aeronautique et de l'Espace (SUPAERO), Toulouse, France in 1999: one in propulsion engineering and one in applied mathematics. In 2003, he received his Ph.D. in applied mathematics from the National Institute of Applied Sciences (INSA), Toulouse, France, for his work at the CERFACS laboratory (Toulouse). He then worked at the Innovative Computing Laboratory at the University of Tennessee until 2006, becoming a Senior Scientist. He now holds an appointment as assistant professor in the Department of Mathematical and Statistical Sciences at the University of Colorado Denver. His research interest is in numerical linear algebra with application in high-performance

Downloaded from hpc.sagepub.com at UNIV OF TENNESSEE on June 21, 2011

computing. He has published his work in SIAM J. Scientific Computing, Numerische Mathematik, SIAM J. Matrix Analysis and Applications, ACM Transactions on Mathematical Software, and International Journal of High Performance Computing Applications (to quote a few).

References

Cuesta-Albertos, J. A. and Wschebor, M. (2003). Some remarks on the condition number of a real random square matrix, *Journal of Complexity*, **19**: 548–554.

Dongarra, J. J. Luszczek, P., and Petitet, A. (2003). The LINPACK benchmark: past, present and future, *Concurrency Computat.: Pract. Exper.*, **15**: 803–820.

Edelman, A. (1988). Eigenvalues and condition numbers of random matrices, *SIAM J. Matrix Analysis and Applications*, **9**(4): 543–560.

Fishman, G. S. (1990). Multiplicative congruential random number generators with modulus 2^β: an exhaustive analysis for $\beta = 32$ and a partial analysis for $\beta = 48$, *Mathematics of Computation*, **54**(189): 331–344.

Higham, N. J. (2002). *Accuracy and Stability of Numerical Algorithms*, second edition, Philadelphia, PA: Society for Industrial and Applied Mathematics.

Hough, D. (1989). Random story, *NA Digest*, **89**(1). http://www.netlib.org/na-digest-html.

Knuth, D. E. (1997). *The Art of Computer Programming*, Vol. 2, third edition Addison-Wesley, Reading, MA.

Schreiber, R. (1989) Hough's random story explained, *NA Digest*, **89**(3). http://www.netlib.org/na-digest-html.

Downloaded from hpc.sagepub.com at UNIV OF TENNESSEE on June 21, 2011