Preprint
RAL-P-2012-009

On the effects of scaling on the performance of Ipopt

JD Hogg, JA Scott

December 2012

Submitted for publication in Computational Optimization and Applications

ISSN 1361- 4762

On the effects of scaling on the performance of Ipopt

J. D. Hogg and J. A. Scott[*]
Scientific Computing Department,
STFC Rutherford Appleton Laboratory,
Harwell Oxford, Didcot OX11 0QX, UK

December 19, 2012

Abstract

The open-source nonlinear solver Ipopt (`https://projects.coin-or.org/Ipopt`) is a widely-used software package for the solution of large-scale non-linear optimization problems. At its heart, it employs a third-party linear solver to solve a series of sparse symmetric indefinite systems. The speed, accuracy and robustness of the chosen linear solver is critical to the overall performance of Ipopt. In some instances, it can be beneficial to scale the linear system before it is solved.

In this paper, different scaling algorithms are employed within Ipopt with a new linear solver `HSL_MA97` from the HSL mathematical software library (`http://www.hsl.rl.ac.uk/`). An extensive collection of problems from the CUTEr test set (`http://www.cuter.rl.ac.uk/`) is used to illustrate the effects of scaling.

1 Introduction

The Ipopt [18] package is designed to solve large-scale non-linear optimization problems of the form

$$\begin{array}{ll} \min_{x\in\mathbb{R}^n} & f(x) \\ \text{s.t.} & g(x) = 0, \\ & x_L \le x \le x_U, \end{array}$$

where $f(x) : \mathbb{R}^n \to \mathbb{R}$ is the objective function and $g(x) : \mathbb{R}^n \to \mathbb{R}^m$ are the constraint functions. Note that more general formulations can be expressed in this form. Ipopt implements a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. This involves solving the indefinite sparse symmetric linear system

$$\begin{pmatrix} W_k + \Sigma_k + \delta_w I & A_k \\ A_k^T & -\delta_c I \end{pmatrix} \begin{pmatrix} d_k^x \\ d_k^\lambda \end{pmatrix} = \begin{pmatrix} b_k^x \\ b_k^\lambda \end{pmatrix} \qquad (1)$$

where δ_w and δ_c are chosen such that the matrix has inertia $(n, m, 0)$. W_k and A_k are the Hessian of the Lagrangian and the Jacobian of $g(x)$, respectively, evaluated at the current trial point. Σ_k is a positive semi-definite diagonal matrix that has values approaching 0 and $+\infty$ as the algorithm converges to a solution. The immediate result of this is that the condition number of the system may increase dramatically as optimality is approached. Further, it is critical that the solution method for (1) accurately reports the inertia of the matrix to allow the Ipopt algorithm to choose δ_w and δ_c. The Ipopt algorithm also uses multiple second-order corrections computed using the same system matrix as in (1) with different right-hand sides, so efficient reuse of information from the initial solution is desirable.

[*]This work was supported by EPSRC grant EP/I013067/1

For these reasons, a sparse direct method is used to solve the system. If, for convenience of notation, we denote the indefinite system (1) by

$$Ay = b, \tag{2}$$

a direct method computes a factorization of the form $PAP^T = LDL^T$, where P is a permutation matrix (or, more generally, a product of permutation matrices) that holds the pivot order and is chosen to try to preserve sparsity and limit growth in the size of the factor entries, L is a unit lower triangular matrix and D is a block diagonal matrix with blocks of order 1 or 2. The solution process is completed by performing forward and back substitutions (that is, by first solving a lower triangular system, a system involving D and then an upper triangular system). The quality of the solution is generally dependent on a stability threshold pivoting parameter u: the larger u is, the more accurate the solution but this can be at the cost of additional floating-point operations and more fill in L (see, for example, Duff, Erisman and Reid [4], page 98). For some factorizations (particularly those where a small u was used), it may be necessary to use a refinement process to improve the quality of the computed solution; normally this is iterative refinement, though more sophisticated schemes are sometimes used [1, 9].

In some applications, matrix scaling can be used to reduce both the computational time and memory requirements of the direct solver while also improving the accuracy of the computed solution, thus limiting the need for iterative refinement and also reducing the number of iterations used by Ipopt. We can apply a symmetric (diagonal) scaling matrix S to (2)

$$SASS^{-1}y = Sb.$$

This is equivalent to solving the following system

$$\hat{A}z = \hat{b}, y = Sz,$$

where $\hat{A} = SAS$ and $\hat{b} = Sb$. The Ipopt "Hints and Tips" webpage states "*If you have trouble finding a solution with Ipopt, it might sometimes help to increase the accuracy of the computation of the search directions, which is done by solving a linear system. There are a number of ways to do this: First, you can tell Ipopt to scale the linear system before it is sent to the linear solver*". How to find a good scaling S is still an open question, but a number of scalings for sparse linear systems have been proposed and are widely used in many different application areas. In this paper, we experiment with scalings available from the HSL mathematical software library [12]. In particular, we examine the use of a number of different scaling algorithms with the new sparse solver `HSL_MA97` [10] within Ipopt. We use a large collection of problems from the CUTEr test set. Our aim is to illustrate the effectiveness of scaling but also to show how it can significantly add to the Ipopt runtime. Our key contribution is a detailed study of the effects of different scaling strategies used with Ipopt that will assist users in making an informed decision about how and when scaling should be tried. We also introduce new heuristics that can be used within Ipopt to limit how often the scaling matrix S is recomputed without increasing the number of Ipopt iterations, thus reducing not only the scaling time but also the total Ipopt runtime.

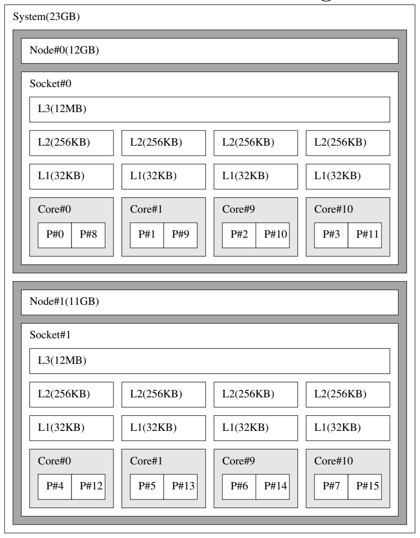
The rest of this paper is organised as follows. We end this section by outlining the experimental setup we will use. In Section 2, we briefly describe the different scaling algorithms that are implemented by routines in the HSL library and will be used in our study. Numerical results for these scalings used with `HSL_MA97` within Ipopt are presented in Section 3. In Section 4, we look at the effect of selectively applying scaling as the Ipopt algorithm progresses. Our recommendations for users are summarised in Section 5 and some final remarks are made in Section 6.

1.1 Test environment

All our experiments are performed using `HSL_MA97` (Version 2.0.0) on the test machine summarised in Figure 1. Although `HSL_MA97` is a parallel code, all reported timings are serial times. We use Ipopt version 3.10.

The CUTEr experiments are run in double precision using the Ipopt CUTEr interface and a development version of the CUTEr tools that supports 64-bit architectures. Of the 1093 CUTEr problems, we discard

Figure 1: Description of test machine.



Machine specification

Processor	2 × Intel Xeon E5620
Physical Cores	8
Memory	24 GB
Compiler	Intel Fortran 12.0.0
	ifort -g -fast -openmp
BLAS	MKL 10.3.0

534 that take less than 0.1 seconds to solve using the HSL solver `MA57` with no scaling (`MA57` was chosen to select the test problems since it is our most widely used modern sparse direct solver). We then discard 173 problems that Ipopt fails to solve with any combination of (HSL) linear solver and scaling tested (note that although, in this paper, we only report on using `HSL_MA97`, we have performed extensive experiments with other HSL linear solvers and the results of these experiments led to the set of discarded problems that we failed to solve). This leaves us with a set of 386 non-trivial tractable problems, and it is to this set we refer to with the phrase "all problems" henceforth. Except as otherwise stated, the default Ipopt settings are used for `HSL_MA97` and CUTEr. In particular, an automatic choice between approximate minimum degree (AMD) ordering and nested dissection (MeTiS) ordering is used. In addition, for each problem, the initial threshold pivoting parameter u is set to 10^{-8}. Note that this Ipopt choice is significantly smaller than the default setting within `HSL_MA97` of 10^{-2}; the use of small u in an optimization context has been discussed, for example, in [6, 7]. For each test problem, a virtual memory limit of 24 GB and time limit of 1000 seconds is placed on each run. Any run exceeding these limits is classified as having failed.

2 The scalings

No Scaling

Many problems can be solved without the use of scaling and, as our numerical experiments will show, this can often result in the fastest solution times (particularly for small problems). The default option within Ipopt (controlled by the parameter `linear_system_scaling`) is not to prescale the linear systems before calling the solver. However, it should be noted that some of the linear solvers that are available for use within Ipopt incorporate scaling algorithms internally and these may be used by default.

`MC19` and `MC30`

`MC19` is the oldest scaling routine available in HSL; it is an implementation of the method described in the 1972 paper of Curtis and Reid [2]. Its aim is to make the entries of the scaled matrix close to one by minimizing the sum of the squares of the logarithms of the moduli of the entries. That is, it scales a_{ij} to

$$a_{ij}\exp(r_i + c_j),$$

with the aim of minimizing the sum of the squares of the logarithms of the entries:

$$\min_{r,c} \quad \textstyle\sum_{a_{ij}\neq 0} \log|a_{ij}| + r_i + c_j.$$

This is achieved by a specialized conjugate gradient algorithm.

MC19 is available as part of the HSL Archive (`http://www.hsl.rl.ac.uk/archive/`). This is a collection of older routines, many of which have been superseded by routines in the main HSL library. In particular, MC19 has been superseded by MC29 for unsymmetric matrices and by MC30 for symmetric matrices. These codes are again based on [2]. The main differences between MC19 and MC30 are that the former uses only single precision internally to compute the scaling factors, while the latter uses double, and MC30 uses a symmetric iteration, while MC19 uses an unsymmetric iteration with the result averaged between rows and columns at the end to obtain symmetric scaling factors.

MC64

Given an $n \times n$ matrix $A = \{a_{ij}\}$, the HSL package MC64 seeks to find a matching of the rows to the columns such that the product of the matched entries is maximised. That is, it finds a permutation $\sigma = \{\sigma(i)\}$ of the integers 1 to n that solves the maximum product matching problem

$$\max_{\sigma} \prod_{i=1}^{n} |a_{i\sigma(i)}|.$$

The matrix entries $a_{i\sigma(i)}$ corresponding to the solution σ are said to be *matched*. Once such a matching has been found, row and column scaling factors r_i and $c_{\sigma(i)}$ may be found such that all entries of the scaled matrix S_rAS_c, where $S_r = \text{diag}(r_i)$ and $S_c = \text{diag}(c_{\sigma(i)})$, are less than or equal to one in absolute value, with the matched entries having absolute value one.

In the symmetric case, symmetry needs to be preserved. Duff and Pralet [5] start by treating A as unsymmetric and compute the row and column scaling r_i and $c_{\sigma(i)}$. They show that if A is non-singular, the geometric average of the row and column scaling factors is sufficient to maintain desirable properties. That is, they build a diagonal scaling matrix S with entries

$$s_i = \sqrt{r_i c_i}$$

and compute the symmetrically scaled matrix SAS. Again, the entries in the scaled matrix that are in the matching have absolute value one while the rest have absolute value less than or equal to one. The symmetrized scaling is included in the package HSL_MC64, a Fortran 95 version of the earlier code MC64, with a number of additional options.

MC77

Equilibration is a particular form of scaling in which the rows and columns of the matrix are scaled so that they have approximately the same norm. The HSL package MC77 uses an iterative procedure [13] to attempt to make all row and column norms of the matrix unity for a user-specified geometric norm $\|\cdot\|_p$. The infinity norm is the default within MC77. It produces a matrix whose rows and columns have maximum entry of exactly one and has good convergence properties. The one norm produces a matrix whose row and column sums are exactly one (a doubly stochastic matrix).

Recently, Ruiz and Uçar [14] proposed combining the use of the infinity and one-norms. Specifically, they perform one step of infinity norm scaling followed by a few steps of one norm scaling (with more steps of the one norm scaling used for their hard test problems). This combination is the MC77 option available within HSL_MA97, with one step of infinity norm scaling followed by three of one norm scaling.

HSL_MC80

If the matched entries returned by MC64 are unsymmetrically permuted onto the diagonal, they provide good pivot candidates for an unsymmetric factorization. To obtain similar benefits in a symmetric factorization, we must instead symmetrically permute these entries on to the subdiagonals and use 2×2 pivots.

Entry a_{ij} is symmetrically permuted onto the subdiagonal by any permutation in which i and j become adjacent in the final order. It is not necessary to apply this condition for all matched entries. In fact, it is sufficient if each row or column participates in a single 2×2 pivot. We therefore construct a subset of the matching such that an index k is only included in a pair once as either a row or column. If this is done prior to the fill-reducing ordering step of the linear solver, an alternative ordering can be found by considering the columns i and j in a pair as a single entity and applying standard reordering techniques. The resultant matching-based ordering is fill-reducing and has large entries on the diagonal or subdiagonal for most rows and columns.

This approach is already implemented as part of the sparse solver PARDISO [15, 16] and has been discussed in the context of Ipopt by Schenk, Wächter and Hagemann [17]. The code `HSL_MC80` represents our implementation of their technique.

The analyse phase of the linear solver `HSL_MA97` optionally computes the matching-based ordering and corresponding scaling by calling `HSL_MC80`. Note that this approach has a number of potential downsides:

- The cost of the analyse phase can be substantially increased.
- The analyse phase may need to be rerun whenever the numerical values change.
- The predicted fill-in will, in most cases, be higher than if numerical values were not taken into account when ordering.

However, for problems that require substantial modifications to the pivot sequence during the numerical factorization to maintain stability, the actual cost of the factorization may be reduced [11]. Note that the scaling used is the same as returned by `MC64`, however the ordering of the matrix is different.

3 Effects of scaling on every iteration

In this section, we look at recomputing and applying the scaling at each Ipopt iteration. Figure 2 shows a performance profile [3] comparison of running Ipopt with each of the scalings discussed in Section 2 and with no scaling. This demonstrates that not scaling leads to the fastest solution times for most of the test problems. However, the test set of 386 problems is skewed towards smaller problems for which, as later results will demonstrate, the scaling time can represent a significant overhead (and can dominate the linear solver time). Figure 3 shows the same profile restricted to the 54 problems that take more than 0.1s (in serial) per iteration. For these problems (which we will refer to as the *large* problems) there is a much weaker differentiation between scaling and not scaling. If we consider the asymptotes of these performance profiles, we obtain a figure on reliability (the percentage of problems satisfactorily solved); these are summarised in Table 1.

Table 1: Reliability of different scalings used on the set of all (non-trivial, tractable) problems and the subset of large problems.

	All problems	Large problems
None	92.8%	94.4%
`MC19`	92.0%	90.7%
`MC30`	92.0%	92.6%
`MC64`	92.5%	96.3%
`MC77`	87.8%	90.7%
`MC80`	92.2%	90.7%

Closer examination of the results reveals that for each of the scalings there is little difference in the number of Ipopt iterations used. For over 75% of the problems, each of the scalings resulted in the same number of iterations being used. For the majority of the remaining problems, `HSL_MC80` led to the smallest

Figure 2: Performance profile of times using different scalings (all problems).

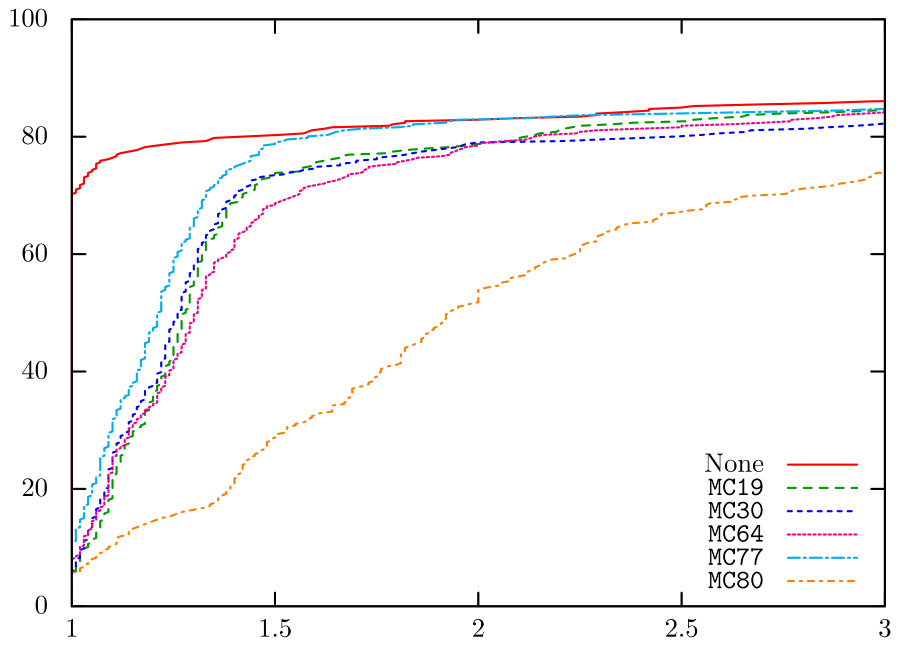


Figure 3: Performance profile of times using different scalings (large problems).

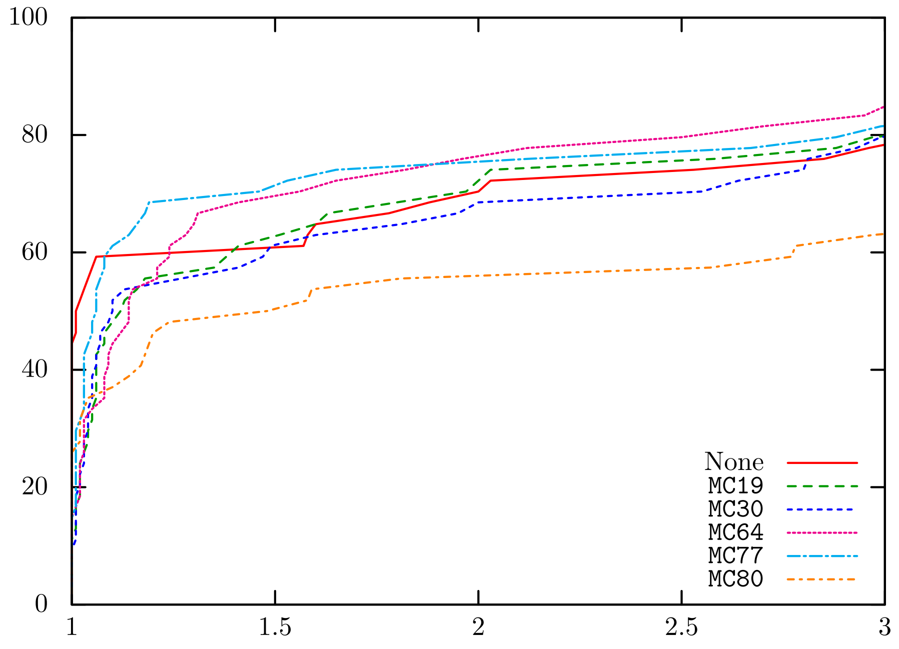


Figure 4: Performance profile of the average number of delayed pivots per factorization for different scalings (all problems).

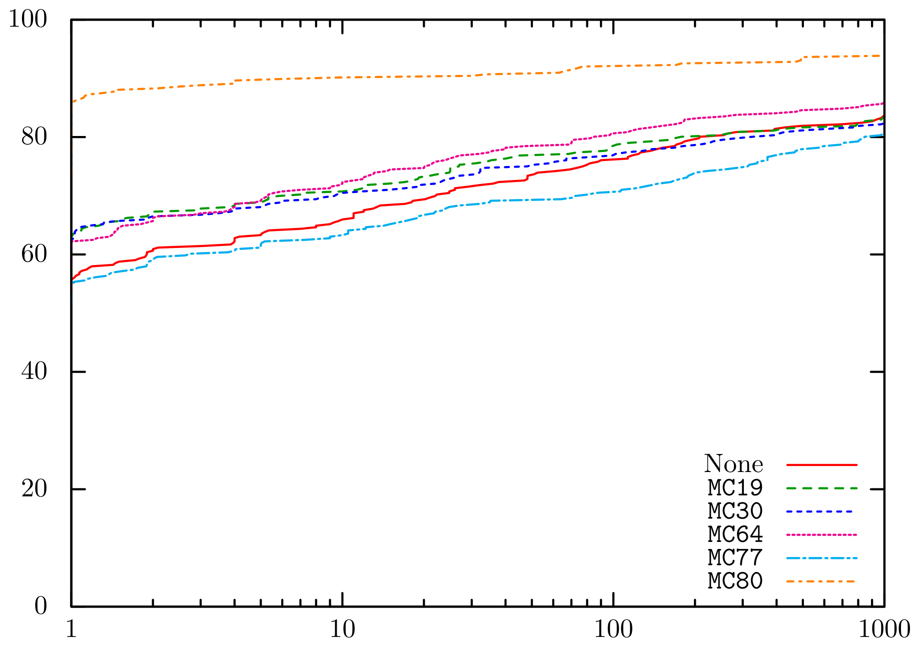
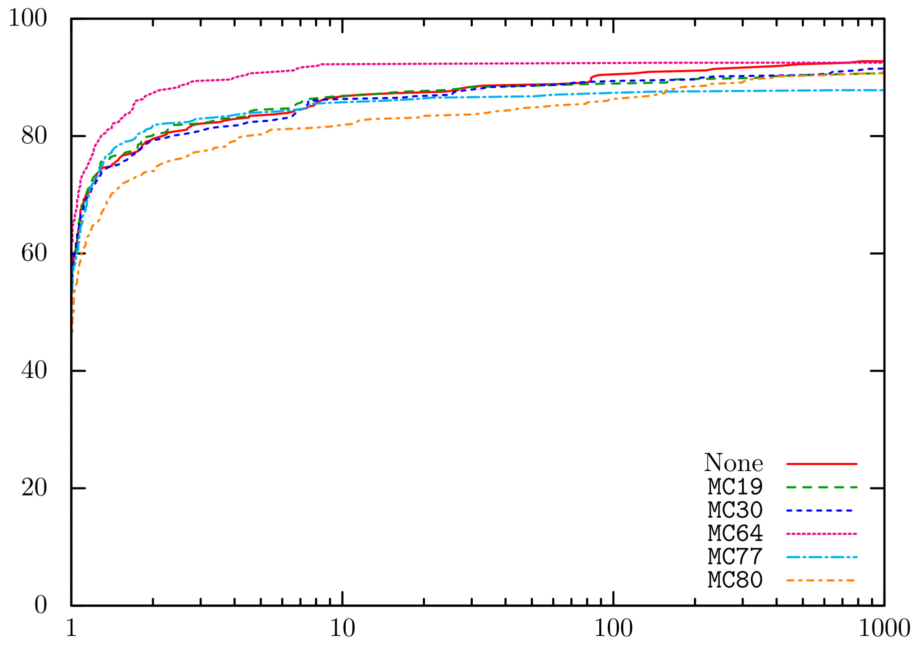


Figure 5: Performance profile of the total factorization floating-point operations for different scalings (all problems).



number of iterations, followed by MC64, with the other scalings only a short distance behind. The main differences in the runtimes are explained by the number of delayed pivots that occur (that is, the number of modifications that must be made to the pivot sequence selected by the analyse phase during the factorization phase to ensure numerical stability) and the number of floating-point operations required to perform the factorizations. Figures 4 and 5 show performance profiles of the average number of delayed pivots per factorization and the total number of floating-point operations, respectively. We observe that while HSL_MC80 is very successful at eliminating delayed pivots, it does so at a substantial increase in the number of floating-point operations required and this leads to it being the most expensive option overall. However, MC64 also produces significantly fewer delayed pivots than the remaining scalings and is able to leverage this to require fewer floating-point operations. The reason MC64 is not the outright best scaling option is that the call to MC64 itself is computationally expensive. Furthermore, we have observed that, if the number of possible maximal matchings is small, the chosen scaling can be poor.

Table 2: Ipopt runtimes (in seconds) for selected problems on which one scaling significantly outperforms the others. A - indicates the run failed (time or memory limit exceeded or failed to converge to a feasible solution). The numbers in brackets are the percentages of the runtime spent on scaling.

Problem	No scaling	MC19	MC30	MC64	MC77	MC80
SPIN	**0.11** (0)	0.14 (23)	0.34 (12)	0.19 (21)	0.15 (13)	0.20 (40)
C-RELOAD	-	**0.95** (19)	-	2.13 (56)	-	2.18 (55)
GRIDNETA	-	0.40 (17)	**0.25** (22)	0.43 (42)	27.8 (29)	0.44 (39)
BRAINPC1	29.9 (0)	-	-	44.2 (80)	**5.88** (6)	-
CBRATU3D	1.52 (0)	1.50 (0.2)	1.50 (0.2)	1.52 (0.2)	1.52 (0.2)	**0.28** (5)
STCQP1	30.5 (0)	709 (0.4)	8.47 (1)	**3.68** (66)	-	7.80 (34)
EIGENC	161 (0)	250 (0.2)	238 (0.3)	171 (4)	317 (0.1)	**68.3** (8)

Table 3: Iteration counts for selected problems using different scalings. Factorization counts are given in () brackets. A - indicates the run failed (time or memory limit exceeded or failed to converge to a feasible solution).

Problem	No scaling	MC19	MC30	MC64	MC77	MC80
SPIN	7 (16)	7 (15)	9 (18)	7 (22)	7 (17)	7 (14)
C-RELOAD	-	111 (242)	-	116 (362)	-	123 (265)
GRIDNETA	-	27 (59)	20 (35)	21 (53)	1471 (4449)	20 (31)
BRAINPC1	82 (171)	-	-	60 (126)	39 (77)	-
CBRATU3D	3 (4)	3 (4)	3 (4)	3 (4)	3 (4)	3 (4)
STCQP1	14 (18)	123 (407)	14 (18)	14 (18)	-	14 (14)
EIGENC	14 (22)	14 (21)	14 (24)	13 (18)	14 (20)	13 (17)

Tables 2–5 report results for a small selection of the test problems that were chosen to illustrate how each of the scalings can outperform the others and to highlight the points of the previous paragraph. Note that a single Ipopt iteration can result in multiple factorizations as it attempts to determine the pertubation required to achieve the correct inertia, or if iterative refinement fails and recovery mechanisms are engaged. For those problems with few delayed pivots and thus little difference in the factorization cost per iteration (SPIN, C-RELOAD, GRIDNETA, BRAINPC1), the scaling that achieves the minimum number of iterations with the smallest scaling time overhead is the fastest. For those problems where delayed pivots are significant (CBRATU3D, STCQP1, EIGNEC), the more expensive MC64 and HSL_MC80 scalings that reduce the number of delayed pivots lead to the best runtimes.

These findings may be compared with previous research by Hogg and Scott [8, 11] into the effects of scalings on a large set of sparse linear systems arising from a wide range of practical applications. They found that for tough indefinite systems HSL_MC80 produces the highest quality scalings, but takes significantly

Table 4: Total factorization floating-point operation counts (in billions) for selected problems using different scalings. The largest number of delayed pivots (in thousands) for a single factorization is given in () brackets. A - indicates the run failed (time or memory limit exceeded or failed to converge to a feasible solution).

Problem	No scaling	`MC19`	`MC30`	`MC64`	`MC77`	`MC80`
SPIN	0.049 (0)	0.046 (0)	1.03 (1)	0.065 (0)	0.052 (0)	0.040 (0)
C-RELOAD	-	0.660 (0.6)	-	0.933 (0.3)	-	0.684 (0.2)
GRIDNETA	-	0.052 (0)	0.031 (0)	0.047 (0)	4.11 (4)	0.030 (0)
BRAINPC1	13.4 (756)	-	-	0.239 (0)	0.151 (7)	-
CBRATU3D	4.44 (7)	4.44 (7)	4.44 (7)	4.44 (7)	4.44 (7)	0.670 (0)
STCQP1	178 (8)	3390 (8)	11.0 (8)	1.49 (0)	-	8.75 (0)
EIGENC	945 (3)	852 (3)	1000 (3)	746 (3)	846 (3)	387 (0)

Table 5: Average scaling time (in milliseconds) per factorization. A - indicates the run failed (time or memory limit exceeded or failed to converge to a feasible solution).

Problem	`MC19`	`MC30`	`MC64`	`MC77`	`MC80`
SPIN	2.2	2.3	2.4	1.1	5.7
C-RELOAD	0.74	-	3.3	-	4.6
GRIDNETA	1.2	1.6	3.4	1.8	5.4
BRAINPC1	6.0	-	281	4.6	-
CBRATU3D	0.87	1.2	1.0	1.1	3.6
STCQP1	6.5	6.8	135	-	148
EIGENC	28	32	343	18	326

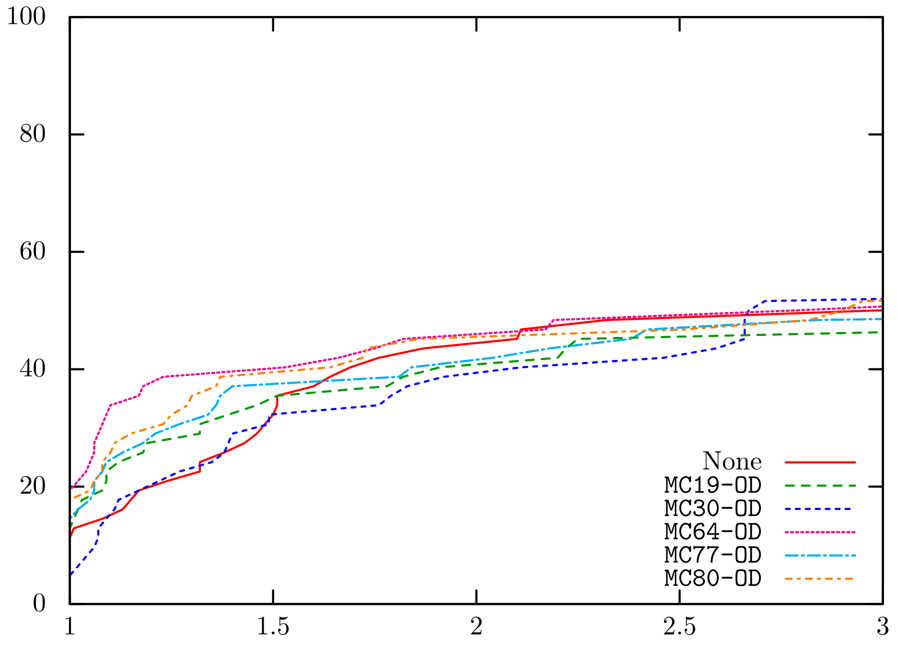
longer than non-matching based scalings to run. In some cases, the time for `HSL_MC80` and `MC64` dominated the cost of the linear solver. Hogg and Scott also reported that for these systems, `MC30` is generally not competitive with `MC77` in terms of quality and run-time.

4 Effects of dynamic scaling

Due to the high overhead of scaling (and particularly of the matching-based scalings), it is logical to experiment with the dynamic use and reuse of scalings. Currently, the default scaling strategy when using an HSL linear solver within Ipopt is to start the computation without scaling and to switch to using scaling when iterative refinement fails to converge to the required accuracy. At this point, to improve the accuracy of subsequent factorizations, the pivot threshold parameter u that is used by the linear solver to control numerical stability while limiting the fill in the computed factors, is increased and all subsequent factorizations use `MC19` scaling. In keeping with Ipopt's configuration parameters, we refer to this as the "on demand" heuristic. Examination of our test set shows that, using `HSL_MA97`, scaling is switched on in this way for only 62 out of the 386 problems (so that, for the majority of the test problems, scaling is not used). This confirms our earlier finding that for the entire test set no scaling is the winning strategy (Figure 2). For the subset of 62 problems, Figure 6 compares the results of using on demand scaling with each of our scalings. When no scaling is used, the failed iterative refinement is resolved by a perturbation of the optimization parameters. We observe that we are able to solve around half of these tough problems.

We have established that the effect of scaling is stronger than merely increased accuracy: it can also be used to reduce the number of delayed pivots (and thus the fill in the factors and the flop count). This suggests the introduction of a second heuristic related to this. Further, it is not clear whether it is necessary to recompute the scaling at *each* iteration, or if we can reuse the scaling computed at an earlier iteration. We therefore modified our `HSL_MA97` Ipopt driver to allow a number of different heuristics to be tested. We use the following notation to denote these:

Figure 6: Performance profile of times for different scalings used with the on demand heuristic (problems that enable scaling only).



`MCXX-OD` "on demand" heuristic, where `MCXX` scaling is computed for subsequent factorizations after the first failure of iterative refinement.

`MCXX-ODR` "on demand reuse" heuristic, as `MCXX-OD`, however the scaling is only computed for the next factorization, and then reused thereafter (until iterative refinement again fails to converge).

`MCXX-HD` "high delay" heuristic, where `MCXX` scaling is computed for all subsequent factorization after the first factorization for which more than $0.05n$ delayed pivots occur.

`MCXX-HDR` "high delay reuse" heuristic, as `MCXX-HD`, however the scaling is only computed on the next factorization, and then reused thereafter. The number of delayed pivots reported by that next factorization is recorded and, if for a subsequent factorization more than $0.05n$ *additional* delayed pivots are encountered, a new scaling is computed for the following factorization.

`MCXX-ODHD` "on demand/high delay" heuristic, where `MCXX` scaling is enabled for all subsequent factorizations if either of the `MCXX-OD` or `MCXX-HD` conditions are met.

`MCXX-ODHDR` "on demand/high delay reuse" heuristic, where `MCXX` scaling is enabled for some subsequent factorizations based on a combination of the `MCXX-ODR` and `MCXX-HDR` conditions.

For the on demand heuristic with `MC64` scaling, Figure 7 illustrates the benefits of reusing the scaling (again for the subset of 62 tough problems). For the large problems in this subset we get some worthwhile savings resulting from the reduction in the total cost of scaling without a deterioration in the factorization quality. For each heuristic, similar behaviour was observed with the reuse version outperforming the version that recomputes the scaling. For the complete test set and the subset of large problems Figures 8 and 9, respectively, compare the different reuse heuristics; reliability results are given in Table 6. These show that using scaling to alleviate the number of delayed pivots significantly increases reliability (by preventing the run aborting because the time or memory limits are exceeded), while the conditions based on failure of iterative refinement also assist with reliability but are less effective. These latter conditions also result in a

Figure 7: Performance profile of times for different `MC64`-based heuristics (problems that enable scaling only).

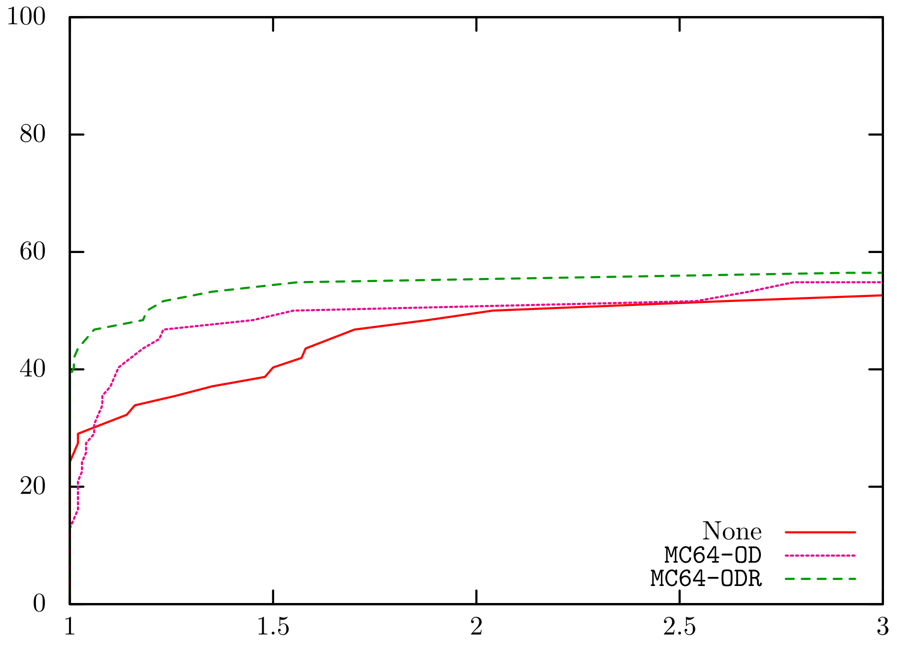


Table 6: Reliability of different scalings used on the set of all (non-trivial, tractable) problems and the subset of large problems.

	All problems	Large problems
None	92.8%	94.4%
`MC64`	92.5%	96.3%
`MC64-OD`	92.2%	94.4%
`MC64-ODR`	92.5%	94.4%
`MC64-HDR`	93.0%	96.3%
`MC64-ODHDR`	93.5%	96.3%

moderate performance advantage for a number of problems. The optimal strategy for the larger problems appears to be `MC64-ODHDR`, while for the entire test set (dominated by small problems), the best heuristic is `MC64-ODR`. This is because the overhead of computing the scaling on the small problems is greater than the saving in the factorization time resulting from reducing the number of delayed pivots.

5 Choice of scaling for specific problem(s)

We recommend that the initial choice of scaling be the `MC64-ODHDR` method; this is the default setting for our `HSL_MA97` driver within Ipopt. If the performance is not satisfactory, the user can enable printing of factorization statistics and information on the use of scaling by selecting the appropriate Ipopt output options.

If there are a large number of delays on the first iteration, an option exists in our driver to force the use of `MC64` on the first iteration in addition to the default heuristic. If a large number of delays occur on iterations that are reusing the scaling computed on an earlier iteration, the user is advised to investigate performance using the `MC64-ODHD` heuristic. Should a large number of delays be present on an iteration

Figure 8: Performance profile of times for different MC64-based heuristics (all problems).

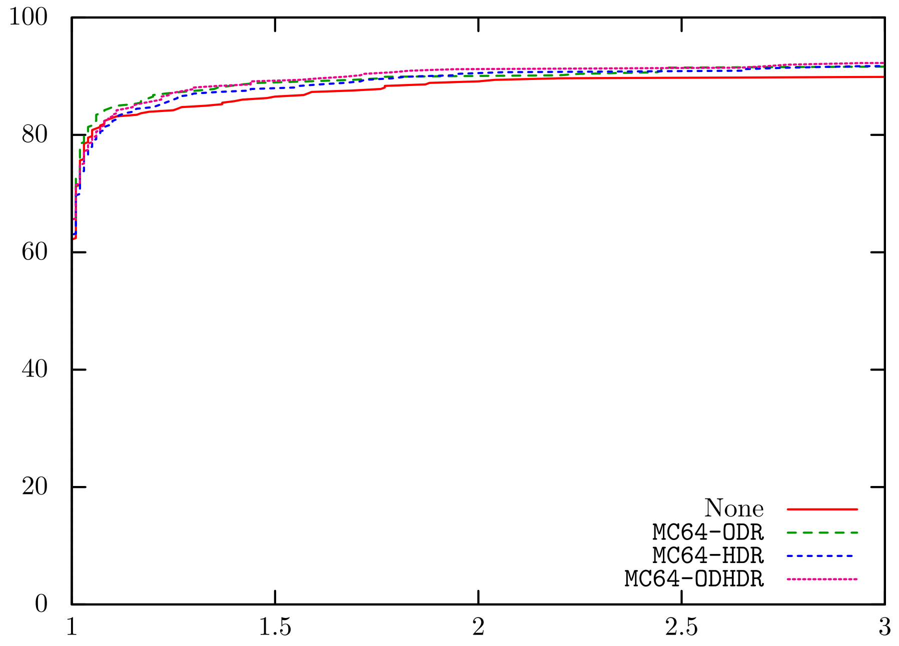
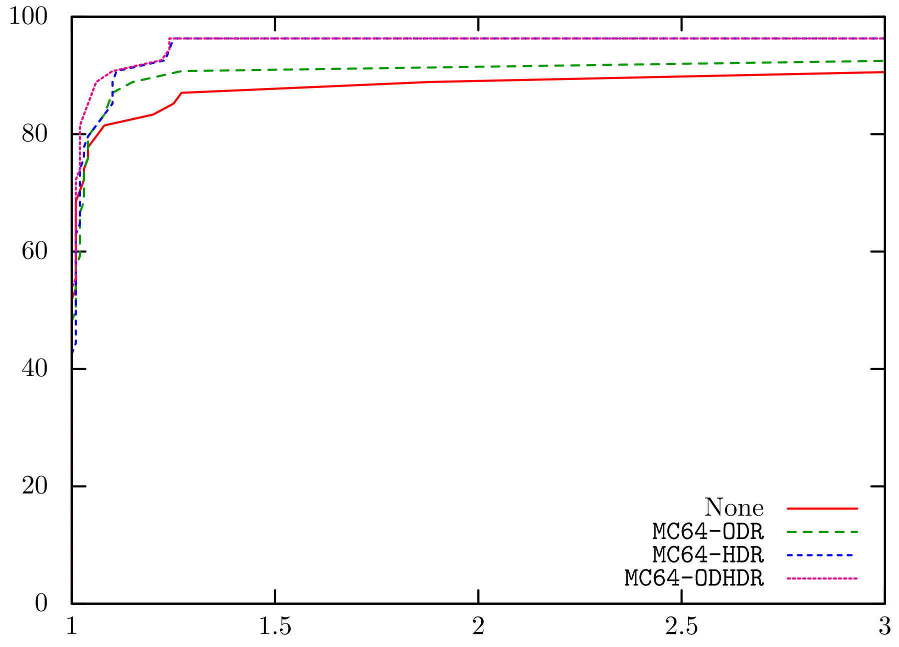


Figure 9: Performance profile of times for different MC64-based heuristics (large problems).



where the MC64 scaling has been computed, use of HSL_MC80 scaling should be considered by enabling the appropriate ordering option.

We have observed that, in some cases where there is a small number of possible maximum matchings across which MC64 can optimize, the computed scaling factors can be poor. If this behaviour is suspected (for example, if a warning has been issued), use of alternative scalings (such as MC77) should be tested. Alternative scalings (or no scaling) may also be considered for small problems for which computing the matching-based scalings can add an unacceptably large overhead.

For cases where top performance is required, we recommend comprehensive testing using all possible scalings (including no scaling) and heuristics that are included in our driver so as to determine the best option for the specific class of problems involved.

6 Conclusions

In this study, we have used the CUTEr test set to explore the effects of scalings within Ipopt. This set is dominated by small problems. For these, the factorization time per iteration is a small proportion of the total Ipopt runtime and so the potential gains from scaling are limited. In particular, an expensive matching-based scaling, while improving reliability, can dominate the total runtime. For larger problems within the CUTEr set, we have seen that the use of scaling can offer worthwhile savings but this is highly problem dependent. Heuristics based, for example, on the time taken to compute the scaling on the first iteration could be considered but time-based heuristics are likely to be very machine dependent. Instead, we have proposed a number of heuristics based on enabling scaling when either iterative refinement fails or the number of delayed pivots becomes large. To reduce scaling costs, we have also considered the reuse of scaling and showed that this can lead to reductions in the runtime. We would welcome the opportunity to run our tests on other large-scale problems, particularly tough problems that are not necessarily well scaled; we are always seeking to expand the set of test problems that we have available.

All our reported timings were for runs performed in serial. HSL_MA97 is, however, a parallel solver. Currently, only serial implementations of the considered scalings are available. If we run Ipopt with HSL_MA97 in parallel, for large problems in particular the cost of the serial scaling can account for a much larger proportion of the runtime, making the reuse of scaling an attractive option. It also suggests that, to avoid scaling being a bottleneck, parallel scaling implementations are needed; this is something we plan to explore in the future.

Based on our findings, we have developed an Ipopt interface for the new HSL linear solver HSL_MA97 that offers the full range of scalings tested in this paper. By including all the heuristics proposed in Section 4, the interface provides users with a straightforward way to experiment to find what works well for their problems. As we have observed that different scalings and different heuristics work best for different problems, we would highly recommend trying out the different options, particularly if many problems of a similar type are to be solved.

Finally, we note that the linear solver HSL_MA97 together with the scaling routines MC30, MC64 and MC77 and ordering/scaling routine HSL_MC80 are included in the 2011 release of HSL. The scaling routine MC19 is part of the HSL Archive. All use of HSL routines requires a licence; details of how to obtain a licence and the routines are available at http://www.hsl.rl.ac.uk/ipopt. Note that routines in the HSL Archive are offered for free personal use to all while those in HSL 2011 are available without charge to academics for their teaching and research. In all cases, a separate commercial agreement is required to allow redistribution of the code and/or the resulting binaries.

Acknowledgements

We are grateful to our colleague Nick Gould for supplying the CUTEr test set and for commenting on a draft of this paper.

References

[1] M. Arioli and I. S. Duff, *FGMRES to obtain backward stability in mixed precision*, Technical Report RAL-TR-2008-006, Rutherford Appleton Laboratory, 2008.

[2] A. R. Curtis and J. K. Reid, *On the automatic scaling of matrices for Gaussian elimination*, Journal of the Institute of Mathematics and Its Applications, 10 (1972), pp. 118–124.

[3] E. D. Dolan and J. J. Moré, *Benchmarking optimization software with performance profiles*, Mathematical Programming, 91 (2002), pp. 201–213.

[4] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, 1986.

[5] I. S. Duff and S. Pralet, *Strategies for scaling and pivoting for sparse symmetric indefinite problems*, SIAM J. on Matrix Analysis and Applications, 27 (2005), pp. 313–340.

[6] R. Fourer and S. Mehrotra, *Solving symmetric indefinite systems in an interior-point method for linear programming*, Mathematical Programming, 62 (19993), pp. 15–39.

[7] P. E. Gill, M. A. Saunders, and J. R. Shinnerl, *On the stability of Cholesky factorization for symmetric quasidefinite systems*, SIAM J. on Matrix Analysis and Applications, 17 (1996), pp. 35–46.

[8] J. D. Hogg and J. A. Scott, *The effects of scalings on the performance of a sparse symmetric indefinite solver*, Technical Report RAL-TR-2008-007, Rutherford Appleton Laboratory, 2008.

[9] ——, *A fast and robust mixed precision solver for the solution of sparse symmetric linear systems*, ACM Transactions on Mathematical Software, 37 (2010). Article 17, 24 pages.

[10] ——, *HSL_MA97: a bit-compatible multifrontal code for sparse symmetric systems*, Technical Report RAL-TR-2011-024, Rutherford Appleton Laboratory, 2011.

[11] ——, *A study of pivoting strategies for tough sparse indefinite systems*, Technical Report RAL-TR-2012-009, Rutherford Appleton Laboratory, 2012.

[12] HSL, *A collection of Fortran codes for large-scale scientific computation*, 2011. `http://www.hsl.rl.ac.uk/`.

[13] D. Ruiz, *A scaling algorithm to equilibrate both rows and columns norms in matrices*, Technical Report RAL-TR-2001-034, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 2001.

[14] D. Ruiz and B. Uçar, *A symmetry preserving algorithm for matrix scaling*, Technical Report INRIA RR-7552, INRIA, Grenoble, France, 2011.

[15] O. Schenk and K. Gärtner, *Solving unsymmetric sparse systems of linear equations with PARDISO*, Journal of Future Generation Computer Systems, 20 (2004), pp. 475–487.

[16] ——, *On fast factorization pivoting methods for symmetric indefinite systems*, Electronic Transactions on Numerical Analysis, 23 (2006), pp. 158–179.

[17] O. Schenk, A. Wächter, and M. Hagemann, *Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale nonconvex interior-point optimization*, Journal of Computational Optimization and Applications, 36 (2007), pp. 321–341.

[18] A. Wächter and L. T. Biegler, *On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming*, Mathematical Programming, 106(1) (2006). 25–57.