# Porting a Sphere Optimization Program from LAPACK to ScaLAPACK

Paul C. Leopardi * Robert S. Womersley†

12 October 2008

**Abstract**

The sphere optimization program `sphopt` was originally written as a sequential program using LAPACK, and was converted to use ScaLAPACK, primarily to overcome memory limitations. The conversion was relatively straightforward, using a small number of organizing principles which are widely applicable to the ScaLAPACK parallelization of serial code. The main innovation is the use of a compressed block cyclic storage scheme to store two symmetric matrices in little more than the storage needed for one matrix. The resulting `psphopt` program scales at least as well as the ScaLAPACK Cholesky decomposition routine which it uses.

# Contents



*Centre for Mathematics and its Applications, Australian National University. `mailto:paul.leopardi@maths.anu.edu.au`

†School of Mathematics and Statistics, University of New South Wales

# 1 Introduction

The sphere optimization program `sphopt` finds a point set on the unit sphere $\mathbb{S}^2 \subset \mathbb{R}^3$ which maximizes the determinant of a certain Gram matrix. The `sphopt` program is limited by the memory size of the computer on which it runs. This paper briefly describes `psphopt`, a parallel version of `sphopt` which uses ScaLAPACK to overcome these memory limitations. An earlier paper [12] describes the optimization problem and its significance in more detail. A draft report [9] provides more detail on the structure of `psphopt`.

Consider the polynomial space $\mathbb{P}_n(\mathbb{S}^2)$, the space of spherical polynomials of degree at most $n$. This is a reproducing kernel Hilbert space of dimension $(n+1)^2$ with reproducing kernel

$$G_n(x,y) := \widetilde{G}_n(x \cdot y) = \sum_{\ell=0}^{n} \sum_{k=-\ell}^{\ell} Y_{\ell,k}(x) Y_{\ell,k}(y) = \frac{n+1}{4\pi} P_n^{(1,0)}(x \cdot y), \qquad (1)$$

where $Y_{\ell,k}$ is a spherical harmonic of degree $\ell$ and order $k$, and $P_n^{(1,0)}$ is a Jacobi polynomial of degree $n$ [11, (4.31)] [12, (2.4)–(2.6)] [2, Section 9.6].

A *fundamental system* is a set of $(n+1)^2$ points of $\mathbb{S}^2$ which exactly interpolates all polynomials of $\mathbb{P}_n(\mathbb{S}^2)$. An *extremal* fundamental system $X = \{x_1, \ldots, x_m\}$ is a fundamental system which maximizes the determinant of the Gram matrix $G$ where $G_{i,j} := G_n(x_i, x_j)$, for $i, j$ from 1 to $m = (n+1)^2$.

Given a polynomial degree $n$, the program `sphopt` finds a fundamental system by maximizing the determinant of the Gram matrix $G$. As noted by Sloan and Womersley [12], this is a non-convex problem with many local maxima.

# 2 The `sphopt` optimization program

The matrix $G$ is symmetric and is non-negative definite as a result of (1). We therefore have $\det G = (\det L)^2$ where $G = LL^T$ is the Cholesky decomposition of $G$. To avoid floating point overflow, `sphopt` maximizes $f = \log(\det G) = 2\log(\det L)$, which is the sum of the logs of the diagonal elements of $L$.

To obtain a local maximum of $f$, `sphopt` uses the L-BFGS-B (bound constrained limited memory BFGS) algorithm as implemented in the L-BFGS-B package [14]. The L-BFGS-B algorithm is based on the limited memory BFGS [10] quasi-Newton optimization method. L-BFGS-B requires the function and gradient value at each step and approximates the Hessian. The optimization follows a typical iterative structure:

**repeat**
    determine a search subspace;
    find the optimal point set in this subspace;
    update the current point set;
**until** *the point set is locally optimal* ;

Finding the optimal point set within a subspace typically involves multiple function and gradient evaluations. In the case of `sphopt`, evaluation of the gradient needs the inverse of $G$, and the the matrix $DG$, the element-by-element derivative of the Gram matrix. Specifically, the gradient $\nabla f$ where $(\nabla f)_{k,i} := \partial f / \partial X_{k,i}$ is computed by

$$\nabla f = 2X \ (DG \bullet G^{-1}),$$

where $X_{k,i} := (x_i)_k$, $k = 1, 2, 3$, $(DG)_{i,j} := \widetilde{G}'_n(x_i \cdot x_j)$, and $\bullet$ is the Hadamard (element-by-element) product. In practice `sphopt` uses spherical polar coordinates, and the gradient with respect to these coordinates is computed using the chain rule.

The `sphopt` program uses LAPACK [1] to compute the Cholesky decomposition (`DPOTRF`), matrix inverse (`DPOTRI`), and matrix multiplication (`DSYMM`) which are needed at every step of L-BFGS-B.

# 3 The ScaLAPACK parallelization of `sphopt`

Two reasons why one would want to parallelize a serial numerical program are to reduce running time and to covercome memory constraints. In 2002, when the calculations for the Sloan and Womersley paper [12] were being performed, the main problem was the memory limitations of available hardware. For the high polynomial degrees which were required, the Gram matrix $G$ and its derivative $DG$ would not fit in main memory.

Since both $G$ and $DG$ are symmetric matrices, only one half of each need be stored. Because both are dense matrices, the storage required for $G$ and $DG$ in IEEE 754 double precision floating point is $8(n+1)^4$ bytes, where $n$ is the polynomial degree. Thus for degree 191, the largest degree treated in the Sloan and Womersley paper [12], `sphopt` would need more than 10 GB.

The solution chosen to meet the memory requirements of `sphopt` was to use distributed memory parallelism via ScaLAPACK [5, 7, 3]. ScaLAPACK is a package for distributed memory parallel linear algebra. It contains distributed memory versions of many of the LAPACK linear algebra routines, including the solution of dense systems of equations, matrix inversion and

eigensystems. The Parallel Basic Linear Algebra Subroutines (PBLAS) component of LAPACK includes matrix-vector and matrix-matrix products. The Basic Linear Algebra Communications Subsystem (BLACS) [8] component enables communication between processes. ScaLAPACK is often implemented via the Message Passing Interface (MPI) [13].

The organizing principles behind the ScaLAPACK parallelization of existing code containing LAPACK calls are those for ScaLAPACK code in general. The goal is to maintain correctness and accuracy while improving speed and allowing larger problems to be solved. ScaLAPACK programs execute the same code in multiple processes. Each process identifies itself via process ID, allowing different processes to take different branches. The input data and results of each ScaLAPACK call are distributed between multiple processes. These processes must communicate and must synchronize before each ScaLAPACK call.

To parallelize an existing program which uses LAPACK, it is therefore necessary to examine each LAPACK or BLAS call, and any Fortran 9X matrix operations and convert each to use the following scheme:

1. Distribute operands; 2. Synchronize; 3. Operate; 4. Distribute results.

To parallelize an existing program which has LAPACK calls inside loops, it is necessary to ensure that all relevant processes execute the loops in synchrony. This requirement is closely related to the need for heterogeneous coherence as described in Dongarra et al. [8, Section D.2]. The safest way to implement loops governed by a varying termination condition is to designate one process as the control process, and send it all data which determines the termination condition. The control process then broadcasts its decision to the other processes. This broadcast per loop keeps the processes synchronized and ensures that all processes execute the same number of loops.

The `psphopt` program therefore runs L-BFGS-B code in the control process only and the program uses the following main optimization loop:

1. The control process broadcasts the current point set $X$;
2. Each process creates its own local parts of $G$ and $DG$;
3. Each process uses ScaLAPACK for Cholesky decomposition (`PDPOTRF` [4]), matrix inverse (`PDPOTRI`) and matrix multiplication (`PDSYMM`) to obtain its own parts of $f$ and $\nabla f$;
4. Each process sends its parts of $f$ and $\nabla f$ back to the control process;
5. The control process calls L-BFGS-B with the new values of $f$ and $\nabla f$;
6. L-BFGS-B calculates a new $X$, or stops.

# 4 Compressed block cyclic storage

ScaLAPACK uses the block cyclic storage scheme [3, Section 4.3] to store matrices. This uses two views of a matrix: the global view and the local (per process) view. In ScaLAPACK, the processes are numbered using a two-dimensional scheme, with each process having a process row number and a process column number, each starting from zero.

The row and column dimensions of a matrix are treated, separately, orthogonally, but in exactly the same way, so it is only necessary to describe one of the two dimensions, say rows.

The concept of a block is common to the global and local views of a matrix. For a given matrix, there is a fixed number of rows in a block, for all row blocks except possibly the last, which may be incomplete. Each row block in the global view is given a row block number starting from zero and is allocated cyclically to a process row. If there are $P$ process rows, we have:

```
ROW BLOCK:   0 1 2 ... P-1 P P+1 P+2 ... 2P-1 2P ...
PROCESS ROW: 0 1 2 ... P-1 0  1   2  ...  P-1  0 ...
ROW CYCLE:   0 0 0 ...  0  1  1   1  ...   1   2 ...
```

In the local view of process row $q$, we have:

```
ROW CYCLE:         0 1 2   3  ... c
LOCAL ROW BLOCK:   0 1 2   3  ... c
GLOBAL ROW BLOCK: q P+q 2P+q ... (c-1)P+q
```

Each global row cycle corresponds exactly to a local row block for each process row, except possibly the last row cycle, which may be incomplete.

We now go back to considering the matrix as a 2D structure in terms of row and column cycles. We assume that we have a square process array and that the row and column block sizes are equal. Therefore all cycles are square, except possibly the last cycle in a row or column.

For a symmetric matrix, in the global view, cycle $(m, n)$ equals cycle $(n, m)$ transposed.

Compressed block cyclic storage stores two symmetric matrices in minimal storage with corresponding global blocks stored on the same process, allowing element-by-element operations to be performed locally.

Because ScaLAPACK routines on symmetric matrices touch only the upper or the lower triangle, at the cycle level, we need only store the upper or lower triangle of cycles. If we have two symmetric matrices, say $A$ and $B$, such that we only use ScaLAPACK routines on $A$, we have the opportunity to save memory on each processor by storing most of $B$ in the untouched triangle of $A$.

Let's choose the upper triangle. We store the upper triangle of cycles of symmetric matrix $A$ according to the usual block cyclic storage scheme. This allows ScaLAPACK to operate directly on matrix $A$. For matrix $B$, we need to store three types of cycle:

1. Square cycles of the strict upper triangle of $B$.
2. Square cycles of the diagonal of $B$.
3. Non-square cycles of the ragged right hand edge of $B$.

We store cycles of type 1, say cycle $(m, n)$ with $m < n$, in the lower triangle of $A$, as cycle $(n, m)$. The cycle is stored as-is — it is *not* transposed. This is essential to ensure that we can perform element-by-element operations on $A$ and $B$ locally.

Cycles of type 2 and 3 must be stored in extra storage. Globally, for type 2, we need one cycle of columns and a full set of rows; for type 3, we need up to one cycle of columns and a full set of rows except for the last cycle, which overlaps with the last cycle of type 2. In both cases, we store the cycle as-is, not transposed.

In the case of `psphopt` we store the matrix $G$ as per $A$ and split the storage of $DG$ as per $B$ above.

Note that the compressed block cyclic storage scheme is not the same as the packed symmetric storage scheme of D'Azevedo and Dongarra [6]. The packed symmetric scheme was not used in `psphopt` because the associated prototype code does not provide a packed version of the matrix inverse function `PDPOTRI`.

# 5 Performance and scalability

The `psphopt` program was originally run on the Australian Partnership for Advanced Computing (APAC) National Facility SC cluster, which was a Compaq AlphaServer SC45 cluster with 127 nodes, each containing four ev68 processors clocked at 1 GHz, and 4 GB to 16 GB of main memory. All of the parallel optimization runs were performed on the APAC SC cluster, with the results reported in Sloan and Womersley [12]. The APAC SC cluster was decommissioned in 2005 and replaced by the AC cluster.

The APAC AC cluster is an SGI Altix 3700 Bx2 cluster containing 1920 Itanium 2 processors, arranged as 30 partitions of 64 processors each. Each partition has between 128 GB and 384 GB of RAM. Each Itanium 2 processor is clocked at 1.6 GHz and has three levels of cache, including 6 MB of level 3 cache. The processors are connected via SGI NUMALink4, which has a rated

MPI latency of less than $2\,\mu$s and bandwidth of 3.2 GB/s in each direction. The NUMALink4 connection is hierarchical. Pairs of processors are connected to memory by a Super Hub (SHub). Eight processors and their SHubs are connected by a pair of routers to form a CR-brick. Four CR-bricks, containing a total of 32 processors, are connected to each other via their routers and connected to higher level routers. There are two more levels of routers connecting groups of 128 processors and 512 processors respectively.

The memory hierarchy of the AC cluster is a great improvement on the memory limitations imposed by the SC cluster, but the scheduler used on the APAC AC cluster still limits the amount of memory per CPU which is available to a job.

Performance and scalability of the `psphopt` program was benchmarked by running jobs on the APAC AC cluster for degrees 31, 44, 63, 90, 127 and 180, using 1, 4, 9, 16, 25, 36, 64 and 100 processors. The polynomial degrees were chosen so that the corresponding numbers of points were close to being successive powers of 2, being 1024, 2025, 4096, 8281, 16384 and 32761 respectively.

All jobs used a block size of $128 \times 128$. The `psphopt` code was compiled using Intel Fortran 9.1 with the ScaLAPACK implementation provided by the Intel Math Kernel Library 10.0.011.

For each degree for each number of processors, the timing test ran ten steps of the L-BFGS-B algorithm, which then used ten or more function and gradient evaluations. For each function and gradient evaluation, walltime was measured on process (0,0) for the creation of the $G$ and $DG$ matrice (TG), for the `PDPOTRF` Cholesky decomposition (TF), for the `PDPOTRI` inverse (TI), and for the total, including these times, synchronization time and the time for `PDSYMM` calculation and broadcast of the gradient (TT). For the single processor jobs, timing was for the corresponding LAPACK code.

For some degrees there were a number of evaluation steps where `PDPOTRF` determined that the matrix $G$ was singular. Timing for these evaluation steps had to be excluded, since in those cases the code recalculates $G$ and does the Cholesky decomposition again, doubling TG and TF. There is also a time penalty for the first evaluation step, due to paging and caching. For each degree for each number of processors, the first evaluation step was therefore excluded.

The timings produced were still highly variable in two cases.

1. The variability in TI and TF for evaluation steps within four-processor jobs was high. For degree 63, the standard deviation of TF was as much as 7 percent of the mean. The variability was likely caused by traffic from other jobs on the same eight-processor CR-brick. Scheduling

these jobs as 8 processor jobs but only using eight processors reduced the standard deviation of TI and TF to less than one percent of the mean in all cases.

2. When this 64-processor and 100-processor jobs were repeated, this revealed a high variability in the average value of TF. For example, for degree 127, one 100-processor job had an average TF of 6.57 s while another had 4.26 s. The cause of this variability is complicated: the PBS scheduler on APAC AC allocates a different set of processors for each job; the Cholesky `PDPOTRF` code uses a large number of point-to-point MPI sends and receives; and the NUMALink4 topology of APAC AC does not necessarily match the processor grid which BLACS assumes, so that processors which are adjacent in terms of process row or process column may actually be linked by a path which traverses as many as six routers. The smallest average TF was reported by the jobs which were scheduled to run across the smallest number of groups of 32 processors which were closest to each other in the NUMALink4 topology. These were the jobs whose timing was used in the results reported here.

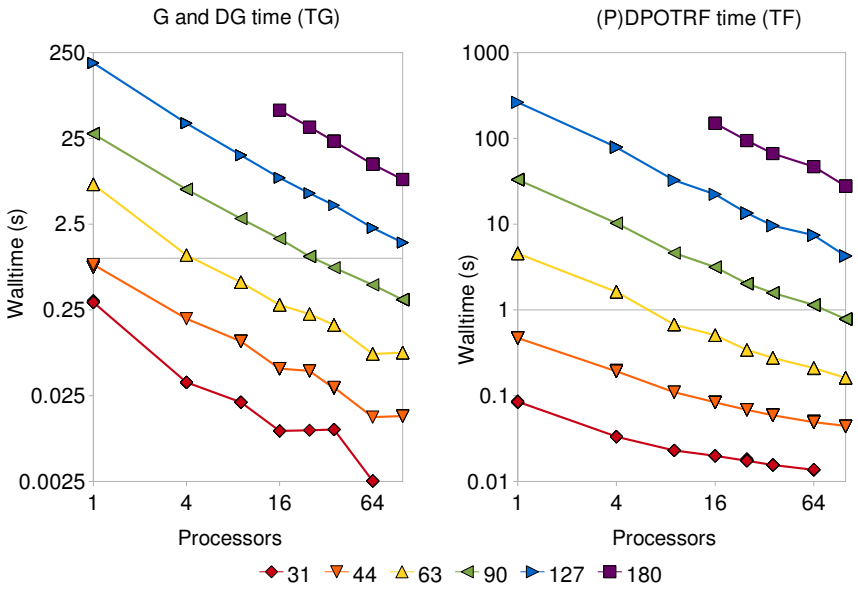


Figure 1: Left: $G$ and $DG$ time (TG). Right: `(P)DPOTRF` time (TF).

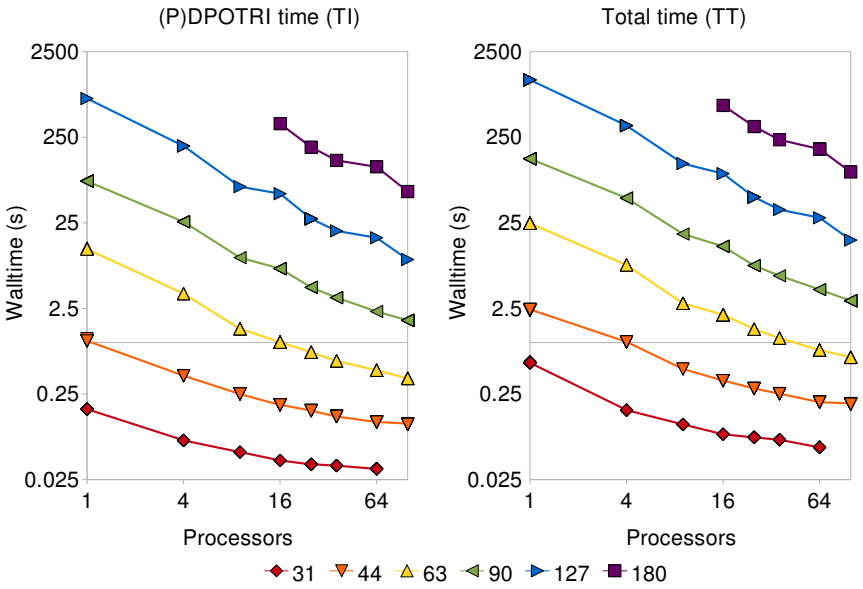


Figure 2: Left: `(P)DPOTRI` time (TI). Right: Total time (TT).

For each polynomial degree the timings were plotted and a log-log regression was performed for walltime versus number of processors, excluding single processor jobs.

Figures 1 and 2 show the timing results for TG, TF, TI and TT respectively. Table 1 displays the scaling exponents obtained from the log-log regression of the timing results. An exponent of -1 indicates perfect strong scalability.

Table 1: Scaling exponents obtained from log-log regression of timing.

| Degree | 31 | 44 | 63 | 90 | 127 | 180 |
|---|---|---|---|---|---|---|
| TG | -0.84 | -0.87 | -0.85 | -0.92 | -0.99 | -1.03 |
| TF | -0.31 | -0.46 | -0.69 | -0.79 | -0.87 | -0.88 |
| TI | -0.28 | -0.41 | -0.67 | -0.81 | -0.89 | -0.90 |
| TT | -0.35 | -0.52 | -0.74 | -0.85 | -0.91 | -0.91 |

These results show that the strong scaling of TG is better than of TI or TF, and that TT also scales a little better than TI or TF. The near horizontal lines in the graphs of TG for degrees 31, 44 and 63 link the cases where the same number of blocks are local to process (0,0).

Memory performance was also tested. The serial program `sphopt` stores $D$ and $DG$ as separate full matrices. The storage needed for double precision for degree 180 is therefore just under 16 GB. The `sphopt` job for degree 180 was run for six hours. It used a maximum of 16 527 MB of virtual memory, matching the estimate. The APAC AC scheduler required the job to request 16 processors even though only one was used. In six hours, the single processor performed just one function and gradient evaluation. In contrast, the 16 processor `psphopt` job performed 12 evaluations in 2 hours 19 minutes, using a total of 10 714 MB spread across the 16 processors.

**Acknowledgements.** The support of the Australian Research Council is acknowledged. Timing computations were performed on the AC cluster of the APAC, under an ANU partner share allocation. The assistance of APAC staff, particularly Margaret Kahn and David Singleton, is gratefully acknowledged. Jörg Arndt read and commented on an early draft of this paper.

# References

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. E. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide (third ed.)*. SIAM, Philadelphia, 1999. `http://www.netlib.org/lapack/lug/`

[2] G. E. Andrews, R. Askey, and R. Roy. *Special Functions*, volume 71 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, 2000.

[3] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, 1997. `http://www.netlib.org/scalapack/slug/`

[4] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. Technical Report 80, LAPACK Working Note, Knoxville, September 1994. `http://www.netlib.org/lapack/lawnspdf/lawn80.pdf`

[5] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra for distributed memory concurrent computers. Technical Report 55, LAPACK Working Note, Knoxville, November 1992. `http://www.netlib.org/lapack/lawnspdf/lawn55.pdf`

[6] E. F. D'Azevedo and J. J. Dongarra. Packed storage extensions for ScaLAPACK. Technical Report 135, LAPACK Working Note, Knoxville, April 1998. http://www.netlib.org/lapack/lawnspdf/lawn135.pdf

[7] J. J. Dongarra, R. A. Vandegeijn, and D. W. Walker. Scalability issues affecting the design of a dense linear algebra library, *Journal of Parallel and Distributed Computing* 22 (3):523–537, September 1994. http://dx.doi.org/10.1006/jpdc.1994.1108

[8] J. J. Dongarra and R. C. Whaley. A user's guide to the BLACS v1.1. Technical Report 94, LAPACK Working Note, Knoxville, May 1997. originally released March 1995. http://www.netlib.org/lapack/lawnspdf/lawn94.pdf

[9] P. Leopardi. Converting a sphere optimization program from LAPACK to ScaLAPACK, 2004. http://wwwmaths.anu.edu.au/~leopardi/conversion-LAPACK-ScaLAPACK.pdf

[10] D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Math. Programming*, 45(3, (Ser. B)):503–528, 1989. http://dx.doi.org/10.1007/BF01589116

[11] M. Reimer. *Multivariate Polynomial Approximation*, volume 144 of *International Series of Numerical Mathematics*. Birkhäuser Verlag, Basel, 2003.

[12] I. H. Sloan and R. S. Womersley. Extremal systems of points and numerical integration on the sphere. *Advances in Computational Mathematics*, 21:107–125, 2004. http://dx.doi.org/10.1023/B:ACOM.0000016428.25905.da

[13] D. W. Walker and J. J. Dongarra. MPI: a standard Message Passing Interface. *Supercomputer*, 12(1):56–68, 1996. http://users.cs.cf.ac.uk/David.W.Walker/MPI/supercomputer96.html

[14] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal. Algorithm 778 L-BFGS-B : Fortran subroutines for large-scale bound constrained optimization. *ACM Transactions on Mathematical Software (TOMS)*, 23 (4): 550–560, December 1997. http://dx.doi.org/10.1145/279232.279236